universidad de león

Escuela de Ingenierías I.I.

Industrial, Informática y Aeroespacial

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

Clasificación en streams

Autor: David Escudero García

Tutor: Miguel Carriegos Vieira

UNIVERSIDAD DE LEÓN
Escuela de Ingenierías I.I.
GRADO EN INGENIERÍA INFORMÁTICA
Trabajo de Fin de Grado

ALUMNO: David Escudero García

TUTOR: Miguel Carriegos Vieira

TÍTULO: Clasificación en streams

CONVOCATORIA: Julio, 2017

RESUMEN:

En este trabajo se pretende examinar el uso de técnicas de aprendizaje aplicadas al análisis de grandes flujos continuos de datos, denominados streams, poniendo énfasis en la tarea de clasificación. En el capítulo 1 se definen y detallan las características de los streams, incluyendo sus aplicaciones, algunas de las técnicas básicas con las que se trabaja y examinando conceptos relevantes en el entorno como el de la deriva conceptual. El capítulo 2 introduce los problemas a la hora de adaptar algunos métodos tradicionales de aprendizaje automático a su uso en streams y presenta algunos de los sistemas propuestos, diseñados específicamente para operar en este entorno, junto a los procedimientos que usan para satisfacer los requerimientos funcionales necesarios. Por último, en el capítulo 3, se implementa en Python, con ayuda de varias librerías, uno de los procedimientos analizados, la DELM, con el propósito de evaluar su rendimiento y compararlo con el de un sistema tradicional como es un perceptrón multicapa.

Palabras clave: stream, clasificación, deriva conceptual, Python

Firma del alumno:	**VºBº Tutor:**

Índice de contenidos

Índice de figuras

Índice de cuadros y tablas

Introducción

En el mundo actual se generan a cada instante enormes volúmenes de información: consultas en motores de búsqueda como Google, transacciones económicas o actividad de red.

Todos estos datos pueden contener conocimiento relevante para tomar mejores decisiones, por lo que su análisis, conocido como minería de datos, es una actividad importante. Como el volumen de información es muy elevado no es práctico estudiarla manualmente, así que esta tarea se lleva a cabo mediante técnicas de aprendizaje automático, que son capaces de aprender de los datos que examinan y generalizar el conocimiento obtenido. Dentro de este campo se encuentra la tarea de clasificación, que consiste en determinar a qué categoría pertenece un nuevo dato. Por ejemplo, determinar si determinada actividad en una tarjeta de crédito constituye un posible fraude.

Un flujo constante y potencialmente infinito de datos se denomina stream, y por sus características no es posible almacenar los datos para su posterior análisis, sino que deben ser procesados en tiempo real. Muchos de los sistemas de clasificación no están pensados para operar en esas condiciones, por lo que es necesario idear nuevos procedimientos que puedan extraer el conocimiento en los datos dando una sola pasada y adaptándose a otra de las características más importantes de los streams: la deriva conceptual. La deriva conceptual constituye un cambio en la distribución de los datos que obliga a revisar el modelo construido por el clasificador o incluso a descartarlo, por lo que la introducción de mecanismos de detección y adaptación al cambio se hacen esenciales. Un ejemplo de deriva conceptual se produce en la detección de spam: las características de los mensajes se modifican para evitar los filtros existentes y el sistema debe poder reconocer esas variaciones para una correcta catalogación.

En este trabajo se pretende profundizar en los conceptos de la clasificación en streams, estudiar las técnicas usadas por algunos de los algoritmos propuestos para su adaptación a este entorno y determinar su rendimiento: tanto en consumo de recursos como en adaptación a la deriva conceptual.

1 Streams

De acuerdo con [43] un stream se puede definir como una secuencia ordenada de datos, potencialmente infinita y que llega continuamente a una velocidad relativamente alta. Por sus características, los sistemas usados para la clasificación en este entorno deben poner especial hincapié en el uso eficiente de recursos.

1.1 Requerimientos y diferencias con el modelo tradicional

En contraposición al modelo tradicional de aprendizaje automático por lotes, los streams tienen varias diferencias:

- El volumen de datos es potencialmente infinito. En el aprendizaje por lotes la magnitud de los datos, aunque puede ser grande, está acotada.
- La velocidad de transmisión de los datos es elevada. En el modelo por lotes la lectura de los datos se produce al ritmo marcado por el algoritmo; en los streams el algoritmo debe adaptarse a la llegada de los datos.
- La llegada de datos se produce de forma secuencial. En el procedimiento tradicional los datos usados están almacenados en un soporte determinado y son completamente accesibles en cualquier momento. En los streams es necesario retenerlos de algún modo si se desea consultarlos posteriormente.
- La susceptibilidad al cambio. Una de las áreas de estudio más importantes en la minería de datos en streams es la adaptación a lo que se conoce como deriva conceptual, que es un cambio en la distribución probabilística de los datos y en los conceptos con los que se trabaja.

Las características descritas se traducen en una serie de requerimientos[5] para los clasificadores diseñados para trabajar sobre streams:

1. Procesar un ejemplo a la vez y revisarlo una vez como máximo. Debido a la introducción secuencial de los ejemplos resulta poco recomendable realizar varias inspecciones del mismo dato, ya que puede causar la pérdida de posteriores elementos. Además la magnitud de la información a examinar hace que tal enfoque no sea computacionalmente eficiente. No obstante este requisito se refiere únicamente a la entrada de datos; es posible que el algoritmo almacene temporalmente algunas entradas para su uso posterior.
2. Usar una cantidad de memoria limitada. La cantidad de datos de un stream no tiene una cota superior por lo que su almacenamiento en memoria es imposible.
3. Trabajar en un tiempo limitado. La velocidad de llegada de los datos es elevada, al igual que el volumen de estos, así que es necesario que el algoritmo sea temporalmente eficiente; al menos lo suficiente como para procesar los ejemplos con más rapidez de la que llegan.
4. Estar preparado para clasificar en cualquier momento. La mayoría de clasificadores tradicionales requieren de una cantidad de tiempo más o menos elevada para entrenar su modelo [1]. A causa de la naturaleza dinámica de los streams el clasificador se suele construir de forma incremental, de modo que se actualiza a medida que procesa los datos. Esto resulta de particular importancia a la hora de enfrentarse a la deriva conceptual, que obliga a ajustar el modelo a nuevos conceptos.

1.2 Deriva conceptual

Además de los requerimientos computacionales, la introducción de la deriva conceptual es uno de los retos fundamentales a los que se enfrenta cualquier sistema de aprendizaje que trabaje sobre streams. El cambio de los conceptos sobre los que se trabaja provoca que el modelo construido hasta el momento vea su precisión reducida o sea erróneo; a lo que hay que responder actualizándolo o realizando una reconstrucción del clasificador usando los nuevos datos.

Desde un punto de vista formal se puede considerar que los ejemplos recibidos son pares (x, y) donde $x = (x_1, x_2, \dots, x_n)$ es un vector de atributos e $y \in \{C_1, C_2, \dots, C_l\}$ es la etiqueta de clase de x. Suponemos que en un instante t los ejemplos son generados por una distribución $P^t(x, y)$ sobre los datos. La deriva conceptual ocurre cuando en un instante $t + \Delta t$ la generación de ejemplos corresponde a otra distribución $P^{t+\Delta t}(x, y) \neq P^t(x, y)$ [25].

En general la deriva conceptual se divide en real y virtual. La deriva real se caracteriza porque se altera $P(y \mid x)$, es decir, varía la clase a la que pertenecen los ejemplos. La deriva virtual puede suponer cambios en la distribución, como por ejemplo $P(x)$, (que determina la proporción entre los diferentes datos) pero estos no afectan a $P(y \mid x)$. En cualquier caso el modelo resulta modificado con la deriva conceptual, sea real o no.

La deriva conceptual también puede definirse según su perfil temporal tal y como se observa en la Figura 1.1.

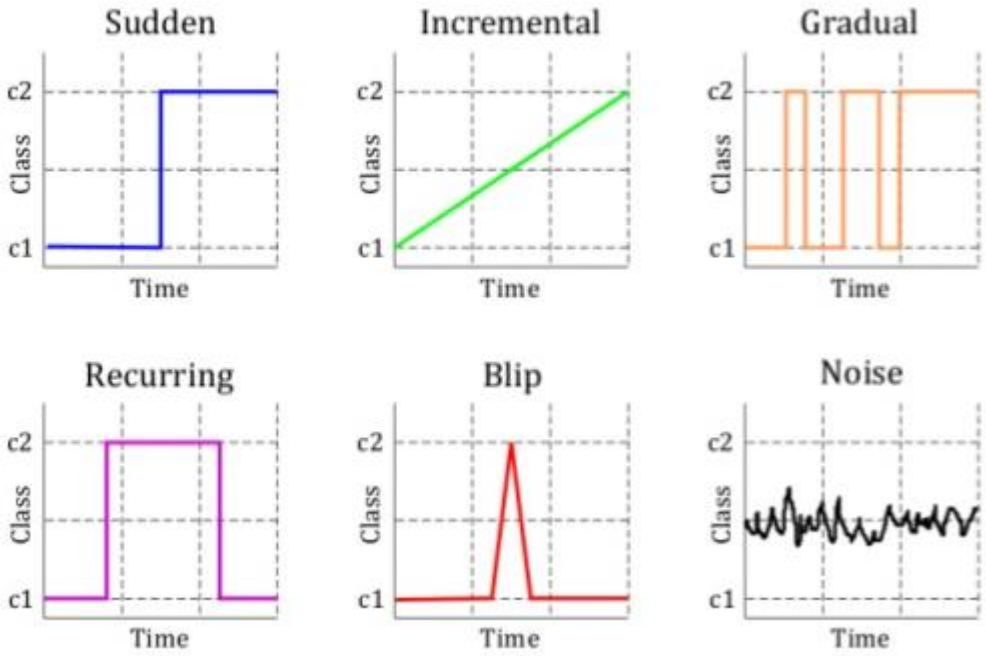

Figura 1.1.- Perfiles temporales de la deriva conceptual [13]

El tipo brusco corresponde a un cambio repentino e irreversible en la distribución de los datos. Los tipos incremental y gradual suelen usarse indistintamente en la literatura y ambos comparten la evolución progresiva, pero técnicamente un cambio gradual implica un cambio en la asignación de clases de los datos, como por ejemplo el cambio en las preferencias musicales

de un usuario. El incremental solo afecta a los valores de los atributos. Una muestra de cambio incremental podría ser el incremento de temperaturas históricas. El cambio periódico implica la aparición repentina de un cambio en la distribución de los datos que desaparece y puede volver a aparecer en el futuro. Un incremento en el interés de los lectores por noticias políticas en período electoral es buen ejemplo de cambio periódico. Un outlier es un dato puntual que se aleja bastante de la distribución actual y no constituye por sí mismo deriva conceptual. Esta ocurrencia se puede corresponder con una intrusión en un sistema crítico. Finalmente, el ruido no es más que una perturbación que no refleja ningún cambio real.

El principal reto de la adaptación de sistemas de aprendizaje automático al cambio se define en el conocido como dilema de la estabilidad-plasticidad. La idea fundamental es que los sistemas deben poder incorporar el nuevo conocimiento al modelo que han creado (plasticidad) pero también es necesario que conserven la información anterior (estabilidad). Esto es un dilema porque demasiada plasticidad provocará el olvido constante de los datos anteriores mientras que un exceso de estabilidad hará que no se añadan los patrones más modernos. Como se verá en el epígrafe 1.3, la mayoría de los algoritmos dan una mayor importancia a los datos más recientes, por lo que se tiende a centrarse más en la plasticidad.

Para la adaptación a la deriva conceptual los sistemas incrementan su grado de plasticidad de forma acorde; sin embargo es posible que la presencia de ruido u outliers sea identificada como un proceso de cambio que provoque la actualización del sistema, aun cuando realmente es información espuria que debería ignorarse. El trabajo en [34] proporciona una distinción formal de la deriva basándose en las ideas de consistencia y persistencia:

La consistencia alude a la magnitud del cambio que se produce entre las diferentes instancias. Siendo θ_t el concepto en el instante t, $\epsilon_t = \theta_t - \theta_{t-1}$ es el cambio en el concepto entre los dos instantes. Si $\epsilon_t \leq \epsilon_c$, siendo ϵ_c un umbral determinado por el usuario, se dice que el concepto es consistente. Esto quiere decir que el cambio detectado entre dos instantes de tiempo tiene que situarse dentro de un intervalo estimado. Según las características de los datos que se analicen el umbral será mayor o menor para reflejar la velocidad del cambio.

La persistencia se refiere a la longitud del intervalo de tiempo durante el que se produce el cambio. Se define X como el tamaño de una ventana de ejemplos, con $\epsilon_{t-p}, \epsilon_{t-p+1}, \ldots, \epsilon_t \leq \epsilon_c$ y $p \geq \frac{X}{2}$. El valor p indica la persistencia del cambio, que es la longitud del intervalo de tiempo durante el que se produce. Por lo tanto, el cambio en un concepto se considera persistente si es consistente a lo largo de un tiempo superior a la mitad del tamaño de ventana considerado. Al igual que en el caso de la consistencia, la persistencia considerada variará en función de la naturaleza de los datos a analizar puesto que depende del tamaño de ventana elegido.

Con estos dos conceptos establecidos se considera que la deriva conceptual es consistente (tiene una magnitud acotada) y persistente (se mantiene en el tiempo), mientras que el ruido no es persistente ni consistente.

1.3 Taxonomía de métodos

En [25] se define una taxonomía de los diferentes algoritmos de clasificación en streams basándose en diferentes características como son la gestión de memoria o el método de aprendizaje que utilizan. Esta clasificación se presenta a continuación.

1.3.1 Memoria

Dentro de este campo se definen dos partes: la gestión de los datos de entrada y el procedimiento de olvido de ejemplos.

Gestión de los datos. Define como el algoritmo se encarga de almacenar los ejemplos de entrada. Debido a la naturaleza dinámica de los streams se considera que los datos más recientes son los más valiosos para la creación de un modelo eficaz, por lo que la gran mayoría de los algoritmos operan almacenando únicamente los últimos ejemplos. Los dos enfoques principales son el uso de uno solo o un conjunto de ejemplos.

Los algoritmos de un único ejemplo no disponen de una memoria de ejemplos en absoluto, simplemente captan la siguiente instancia, la procesan y se deshacen de ella. Este procedimiento va fuertemente asociado al aprendizaje online, que actualiza el modelo dato a dato en función de los errores.

La alternativa es el almacenaje de varios ejemplos. Este sistema mantiene una ventana de los últimos datos que han llegado y a partir de ella crean el modelo de decisión. El algoritmo CVFDT[29] sigue este enfoque. El principal problema al que se hace frente en este caso es determinar cuál es el volumen adecuado de almacenaje; una ventana más grande consumirá más memoria y se adaptará peor a los cambios rápidos, pero a cambio dispondrá de más ejemplos para construir un modelo más preciso.

La elección de un tamaño de ventana adecuado es uno de los principales ejemplos del dilema de estabilidad-plasticidad. Como solución a la disyuntiva se han desarrollado procedimientos para disponer de un tamaño de ventana dinámico, que varía en función de la tasa de cambio de los datos. Un ejemplo de este enfoque aplicado a las SVM se halla en [32].

Además de los dos anteriores, existe el procedimiento de memoria completa. En este caso no se almacenan explícitamente las instancias, por lo que en cierto modo podría considerarse como parte del sistema de único ejemplo. Sin embargo, se mantienen estadísticas relevantes a los datos recibidos para su uso en el modelo. Por ejemplo, ANNCAD[33], basado en el método de los vecinos más cercanos, guarda información sobre la distribución de clases en el espacio de representación.

Procedimiento de olvido. Como la capacidad de memoria es limitada y es necesario adaptarse a cambios hay que definir un método por el cual eliminar aquellos ejemplos desfasados en el modelo actual.

Una opción es el olvido abrupto. Se usa típicamente junto a ventanas de ejemplos y consiste simplemente en desechar el subconjunto más antiguo de la ventana y sustituirlo por datos más recientes. Comúnmente la eliminación se lleva a cabo como una cola FIFO (*First In First Out*): si llega un nuevo ejemplo y la ventana está llena se elimina el más antiguo y se introduce el nuevo.

Por otra parte, los mecanismos de olvido gradual no descartan ejemplos de forma explícita, sino que usan un factor de olvido que disminuye la importancia de los datos antiguos en el modelo actual. La aplicación de este enfoque se asocia principalmente al uso de la gestión de datos de memoria completa y no al almacenaje directo de estos, ya que el volumen de datos de un stream es potencialmente infinito.

Algunos algoritmos no incluyen ningún procedimiento de olvido explícito y simplemente descartan el modelo actual cuando deja de ser válido y crean otro. Un ejemplo de este procedimiento lo sigue la DELM[47].

1.3.2 Detección del cambio

La gran mayoría de sistemas de aprendizaje se actualizan constantemente, por lo que pueden adaptarse a la deriva conceptual de forma gradual. Sin embargo, para poder hacer frente a los cambios súbitos en los datos se suele hacer uso de métodos de detección de cambio. Estos se dividen en tres familias:

- Análisis secuencial. Estos métodos se basan en detectar cuando una variable tiene un valor notablemente distinto al esperado, es decir, a su media. El test de Page-Hinkley[38] es un ejemplo de esta familia.
- Control estadístico de procesos. Estos métodos tienen su origen en los procedimientos de control de calidad de procesos industriales y detectan cambios poco probables modelando el error como una prueba de Bernoulli, que es básicamente un experimento aleatorio en que solo existen 2 resultados: éxito y fracaso. DDM[22] es un ejemplo de esta clase.
- Comparación de dos ventanas de ejemplos. Estos métodos usan una ventana de referencia sobre el total de ejemplos y otra sobre los más recientes. Cuando las distribuciones de los datos difieren entre ellas se considera que se ha producido un cambio al que hay que adaptarse. Un representante de este enfoque es ADWIN[4].

Una discusión en más profundidad de los ejemplos mencionados se lleva a cabo en la sección 1.4.

1.3.3 Aprendizaje

Otro criterio para separar los algoritmos existentes está en el procedimiento que siguen para generalizar un modelo a partir de los datos de entrada.

Modo de aprendizaje. Determina el procedimiento para actualizar el modelo al recibir ejemplos. Los modelos de reentrenamiento e incremental son los principales.

El reentrenamiento consiste en descartar el modelo actual al recibir nuevos datos y crear uno nuevo. Este nuevo modelo suele construirse combinando datos antiguos almacenados en una ventana y los nuevos. Este procedimiento se suele usar para adaptar algoritmos de aprendizaje por lotes para su uso en streams, aunque también se aplica asociado a sistemas de detección de deriva como el DDM. Un ejemplo de reentrenamiento se perfila en [19] para las SVM.

El modelo incremental, por el contrario, actualiza el modelo actual con las nuevas instancias en vez de reconstruirlo desde cero. Dentro de los modelos incrementales se puede destacar el online, en el que la actualización se produce solo si se produce un error en la clasificación de la nueva instancia; y el modelo

de streaming, que simplemente añade las restricciones ya vistas en la sección 1.1.

Métodos de adaptación. Como ya se ha visto, es necesario que los modelos creados deben adaptarse a los posibles cambios en los datos de entrada. Los métodos de adaptación determinan de qué modo, informado o ciego, realizan la adaptación.

El método de adaptación ciego se acomoda a los cambios de forma preventiva, se haya producido cambio realmente o no. El modo de aprendizaje incremental es ciego ya que se actualiza constantemente con los datos de entrada. Otro procedimiento se basa en el uso de ventanas que almacenan los ejemplos más recientes, que son con los que el modelo se mantiene congruente al usarlos para reentrenar.

Los métodos de adaptación informados usan algún mecanismo de detección de cambio como los mencionados en la sección 1.3.2 para determinar cuándo deben actuar para reajustar el modelo.

El reajuste del modelo consiste generalmente en descartar el anterior y construir uno nuevo a partir de datos recientes. La sustitución puede ser global, en la que se descarta completamente el modelo; o local, en la que solo se intercambia una parte. Este último caso se da por ejemplo en el algoritmo CVFDT[29], en el que se una partición de un nodo puede sustituirse por un nuevo subárbol.

1.4 Detección de la deriva conceptual

Dada la importancia del fenómeno de la deriva conceptual en los streams, una amplia variedad de métodos han sido diseñados para su detección, a fin de poder adaptar los modelos de los clasificadores usados en el momento oportuno.

En general, los mecanismos de detección suelen aplicarse al análisis de los cambios en el error del clasificador. Se estima que un clasificador, al incrementar el número de ejemplos analizados, ve su error reducido, o, al menos, estabilizado[23]. Por tanto un cambio significativo en el error calculado corresponde a datos que no se acomodan al modelo actual, es decir, se produce deriva conceptual.

1.4.1 Test de Page-Hinkley

Uno de los métodos es el test de Page-Hinkley[38], que detecta cambios en una variable mediante la desviación respecto a su media. El Algoritmo 1.1 define el procedimiento.

$\overline{x_N}$ es la media de la variable controlada hasta el momento actual. m_N mide la desviación acumulada de la variable respecto de su media hasta el momento actual. El hecho de que no se incluya ningún valor absoluto en el cálculo de m_N se debe a que la variable controlada es el error de clasificación, y solo es relevante el incremento de este, como ya se ha comentado al inicio de la sección. El umbral δ se incluye en el cálculo para tener en cuenta únicamente las desviaciones significativas; es casi imposible que los valores de la variable sean idénticos a su media por lo que si no se tuviese en cuenta δ, el algoritmo acabaría detectando una desviación en la variable tras un determinado número de ejemplos, existiese o no dicha desviación.

Algoritmo 1.1.- Test de Page-Hinkley [42]

Test de Page-Hinkley	
Entrada:	$x_1, x_2, \ldots, x_n$ valores de la variable controlada δ umbral de magnitud λ umbral de error
Salida:	t_d instante en el que se detecta la desviación

1: **Para** n > 0
2: $\overline{x_N} = \frac{1}{N} \sum_{i=1}^{N}(x_i)$
3: $m_N = \sum_{i=1}^{N}(x_i - \overline{x_N} - \delta)$
4: $M_N = \min(m_N, M_N)$
5: **Si** $m_N - M_N > \lambda$
6: Devolver t_d

M_N mantiene el valor mínimo de m_N registrado hasta el momento. Para determinar la existencia de un cambio de determina si la diferencia entre m_N y M_N supera al umbral de error λ. El valor de λ determina la sensibilidad del algoritmo al cambio: un valor demasiado pequeño provocará falsos positivos, pero uno demasiado elevado no detectará cambios con demasiada eficacia.

1.4.2 DDM

Otro sistema es el DDM[22], que determina la presencia de deriva conceptual analizando la evolución de la probabilidad de error. DDM modela el proceso de clasificación como una prueba de Bernoulli y por tanto determina el error como una variable aleatoria binomial. El trabajo aprovecha el hecho de que para un número de ejemplos n (con $n > 30$ en la referencia) la distribución binomial del error es aproximada por una distribución normal con la misma media y varianza.

La transformación en una distribución normal es importante porque su función de densidad de probabilidad es simétrica, lo que permite definir fácilmente una serie de intervalos de confianza para determinar si se produce deriva conceptual.

Los intervalos de confianza definidos son tres:

- Normal: $p_i + s_i < p_{min} + \alpha s_{min}$. No hay cambios detectados.
- Aviso: $p_i + s_i \geq p_{min} + \alpha s_{min}$. Puede estar produciéndose deriva conceptual, pero se espera a la llegada de más ejemplos para confirmarse.
- Deriva conceptual: $p_i + s_i \geq p_{min} + \beta s_{min}$. Se confirma la ocurrencia de la deriva conceptual.

Al igual que con el test de Page-Hinkley, solo se considera relevante el incremento de la variable medida (el error).
Los pasos del método se definen en el Algoritmo 1.2. Para cada nuevo ejemplo se calcula p_i y s_i que denotan la probabilidad de error del clasificador hasta el momento y su desviación típica respectivamente. p_{min} y s_{min} almacenan los mínimos valores registrados para las variables anteriores y se usan como referencia para determinar la ocurrencia de deriva conceptual.

El algoritmo determina en qué situación se encuentra la evolución del error y obra en consecuencia. En el caso de que se pase al estado de aviso los ejemplos entrantes se almacenan en una ventana W. Si se pasa desde el estado de aviso al normal se considera que produjo una falsa alarma y se descarta la ventana. Si se confirma la ocurrencia de deriva conceptual entonces se descarta el clasificador actual y se crea uno nuevo con los ejemplos almacenados en la ventana a fin de que el nuevo modelo esté ajustado a la nueva distribución de datos; además se reinicia el estado de todas las variables.

Algoritmo 1.2.- DDM [25]

DDM

Entrada: C clasificador
S stream de ejemplos (x, y)
α coeficiente de confianza
β coeficiente de confianza

1: $W = \emptyset$
2: estado = normal
3: **Para** todo $x_i \in S$
4: Predecir etiqueta de x_i
5: Actualizar la probabilidad de error p_i
6: $s_i = \sqrt{\frac{p_i(1-p_i)}{i}}$
7: **Si** $p_i + s_i < p_{min} + s_{min}$
8: $p_{min} = p_i$ y $s_{min} = s_i$
9: **Si** $p_i + s_i < p_{min} + \alpha s_{min}$
10: **Si** estado == aviso
11: $W = \emptyset$
12: estado = normal
13: **Si no Si** $p_i + s_i \geq p_{min} + \alpha s_{min}$
14: estado = aviso
15: **Si no Si** $p_i + s_i \geq p_{min} + \beta s_{min}$
16: Reentrenar C con W
17: $W = \emptyset$
18: estado = normal
19: $p_{min} =$ y $0\ p_i = 0$ y $s_{min} = 0$ y $s_i = 0$
20: **Si** estado == aviso
21: $W = W \cup \{x_i\}$

Las constantes α y β sirven para determinar la probabilidad de que un ejemplo se sitúe dentro del intervalo definido. α se define como 2 y β como 3, lo que da una confianza de 95% y 99% respectivamente. Esto se traduce en que siguiendo la distribución normal, para un ejemplo, hay un 95% de probabilidad de que $p_i + s_i < p_{min} + 2s_{min}$. Si esta desigualdad no se cumple quiere decir que con una probabilidad de un 95% el ejemplo procede de otra distribución probabilística, lo que significa la ocurrencia de deriva conceptual de acuerdo a la definición establecida en la sección 1.2.

1.4.3 ADWIN

El algoritmo ADWIN[4] es un método basado en el uso de una ventana deslizante de tamaño variable. Los ejemplos se almacenan en una ventana y se considera que cuando las medias de los elementos de dos subventanas "suficientemente grandes" tienen valores "suficientemente distintos" se puede considerar que el valor esperado de la variable ha cambiado en el tiempo (se ha producido deriva conceptual) y se descarta la porción más antigua de la ventana. El Algoritmo 1.3 describe el funcionamiento de ADWIN.

Algoritmo 1.3.- ADWIN [4]

ADWIN

Entrada: $x_1, x_2, \dots, x_n$ valores de la variable controlada
δ umbral de confianza

1: $W = \emptyset$
2: **Para** n > 0
3: $W = W \cup \{x_n\}$
4: **repetir**
5: eliminar el elemento más antiguo de W
6: **hasta** $|\hat{\mu}_{M0} - \hat{\mu}_{M1}| \geq \epsilon$ sea cierto para toda partición $W = W_0 . W_1$

$\hat{\mu}_{M0}$ y $\hat{\mu}_{M1}$ denotan las medias de los elementos de las subventanas W_0 y W_1 respectivamente. El algoritmo determina los tamaños "suficientemente grandes" y las medias "suficientemente distintas" a partir de la condición de la línea 6.

ϵ se define de la forma siguiente:

$$\epsilon = \sqrt{\frac{1}{2m} . \ln\left(\frac{4}{\delta'}\right)} \quad \text{con} \quad m = \frac{1}{\frac{1}{n_0} + \frac{1}{n_1}} \quad \text{y} \quad \delta' = \frac{\delta}{n}$$

n denota la longitud de la ventana W y n_0 y n_1 las longitudes de las particiones W_0 y W_1 respectivamente. Por supuesto $n = n_0 + n_1$

El cálculo de ϵ se basa en la desigualdad de Hoeffding (tratado en más profundidad en la sección 2.1.1), por lo que está demostrado formalmente que si la diferencia de la medias estimadas de las particiones superan el umbral, se ha producido una variación de la media real de la variable con probabilidad de error δ. Como en el cálculo de ϵ se tiene en cuenta el tamaño de las subventanas mediante m también se cumple con la necesidad de evaluar ventanas "suficientemente grandes".

El algoritmo solo tiene una ventana de ejemplos que se particiona en diferentes puntos para obtener las 2 subventanas que se comparan. Se estima la diferencia de las medias de los datos de todas las posibles subventanas y se elimina el dato más antiguo de la ventana principal si la diferencia supera el umbral establecido. El descarte de elementos se realiza para mantener en la ventana únicamente los elementos procedentes de la nueva distribución. El elevado número de particiones posibles hace que el cálculo pueda ser muy costoso.

En ADWIN, al contrario que en el test de Page-Hinkley, se considera el uso de un valor absoluto en el cálculo de la desviación de la media. Esto es porque la

variable medida en ADWIN no es el error de clasificación sino el valor de un dato. Esto es relevante si se considera que la deriva conceptual incremental descrita en la sección 1.2 viene dada por un cambio en los valores de los datos, por lo que tanto el incremento como la disminución son relevantes. El procedimiento usado implica que será necesario una ventana para cada uno de los atributos de los ejemplos para poder detectar los cambios eficazmente.

1.5 Evaluación

Al igual que con los sistemas de aprendizaje automático que trabajan sobre lotes de datos, los algoritmos aplicados a streams también deben evaluarse para obtener su rendimiento. No obstante además de la precisión de la clasificación es necesario determinar el consumo de recursos y la adaptación a la deriva conceptual.

1.5.1 Evaluación del consumo de recursos

El consumo de recursos se divide en el tiempo de ejecución y el consumo de memoria.

La medición del tiempo de ejecución es directa, y suele expresarse según el número de ejemplos procesados. Se desea que la relación entre el tiempo de ejecución y las instancias procesadas sea lineal, de modo que el procesamiento de cada instancia se realice en un tiempo constante.

El consumo de memoria también debe estar acotado y se suele perfilar determinando cuanta memoria física necesita el sistema a lo largo de toda la ejecución. En el caso ideal un algoritmo necesitaría una cantidad de memoria constante durante todo su funcionamiento; mientras que un tiempo de ejecución lineal se considera adecuado, en el caso del espacio de almacenamiento es inaceptable debido al volumen de datos potencialmente infinito de los streams. En [6] se introduce una métrica denominada horas-RAM. Una hora-RAM equivale a un GB de memoria RAM desplegado durante una hora. Esta métrica está basada en los costes de alquiler de computación en la nube y debido a su magnitud tiene su aplicación para los sistemas que operan analizando volúmenes muy grandes (del orden de Terabytes) de datos.

1.5.2 Estrategias para la evaluación del rendimiento

La realización de test requiere de una organización para determinar cuándo van a realizarse y los datos que van a ser utilizados; una serie de estas estrategias se presentan a continuación.

1.5.2.1 Holdout

El procedimiento holdout es un tipo de validación cruzada. En el entorno de streams el uso de otros procedimientos de validación cruzada, que requieren varias pasadas sobre los mismos datos resulta ineficiente y además incumple el primer requerimiento definido en la sección 1.1; por lo que se recurre a una partición simple del conjunto de datos, creando un grupo de datos de test separados que se usarán para evaluar el modelo periódicamente. Si existe deriva conceptual, realizar una división de los datos de este modo puede no ser adecuado, ya que cabe la posibilidad de que los datos de test seleccionados no reflejen los conceptos actuales con los que trabaja el modelo.

1.5.2.2 Prequential

El modelo prequential o test-then-train se basa en usar cada uno de las instancias del stream primero como dato de test para evaluar el modelo y luego para actualizarlo. También se puede aplicar para bloques de datos y no solo para ejemplos separados. Tal y como se menciona en [25], este enfoque permite el máximo aprovechamiento de los datos al no crear un conjunto separado de test y además realiza los test sobre datos que el modelo no ha visto antes. Este sistema es el más usado para la evaluación de los modelos aplicados a streams. Se esquematiza el procedimiento en la Figura 1.2.

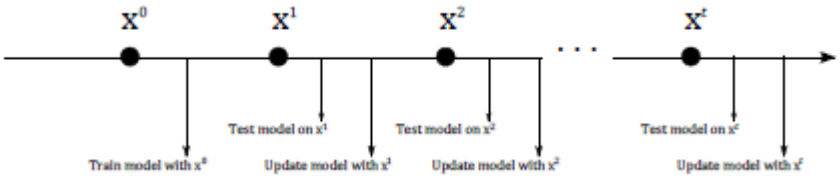

Figura 1.2.- Método de evaluación prequential [43]

En [24], se menciona que este procedimiento es pesimista, por lo que, en las mismas condiciones, el error estimado será mayor que el de otros procedimientos como el holdout. Esto se debe a que el error determinado por el sistema prequential se verá influido por los primeros test realizados, en los que el modelo habría sido entrenado con pocos datos. La solución propuesta pasa por usar un factor de envejecimiento, de modo que el error medio pasa a calcularse como:

$$E_i = \frac{L_i + \alpha E_{i-1}}{n_i + \alpha B_{i-1}} \quad \text{con} \quad 0 \ll \alpha \leq 1 \quad \text{y} \quad B_i = \alpha B_{i-1}$$

α es el factor de envejecimiento; B el número de ejemplos procesados hasta el momento; L_i el error para el nuevo bloque de ejemplos y n_i el número de ejemplos del nuevo bloque.

El resultado de este sistema es que el impacto de los errores más antiguos se ve reducido, por lo que el resultado estimado tendrá en cuenta el modelo en un estado estable, en el que haya procesado más ejemplos.

1.5.2.3 Permutaciones controladas

Otro método se propone en [49], denominado permutaciones controladas. En este trabajo se afirma que el sistema prequential, al realizar una sola pasada sobre los datos, no ofrece suficiente información sobre el rendimiento del modelo creado. En el ámbito de los streams, los modelos no solo deben ser precisos sino que también tienen que poder adaptarse a los cambios en la distribución de los datos. Los test del procedimiento prequential solo proporcionan información sobre la capacidad del modelo para adaptarse a las variaciones que aparecieron en los datos usados, con una velocidad de cambio determinada, en unos instantes de tiempo concretos, pero no indican la capacidad de generalizar ese conocimiento adquirido a otros perfiles de deriva conceptual.

La solución propuesta pasa por realizar diferentes tipos de permutaciones sobre los datos, alterando su orden, pero manteniendo próximos los ejemplos que se encontraban cercanos en la secuencia original a fin de conservar la distribución local.

Se distinguen 3 tipos de permutación que se ilustran en la Figura 1.3:

- Temporal: Los datos se dividen en bloques en puntos aleatorios y estos bloques se disponen en orden inverso al original.
- De velocidad: Se toman una serie de ejemplos de al azar y se colocan al final de la secuencia.
- De forma: Se determina un ejemplo y se intercambia su posición con la de un vecino a su derecha.

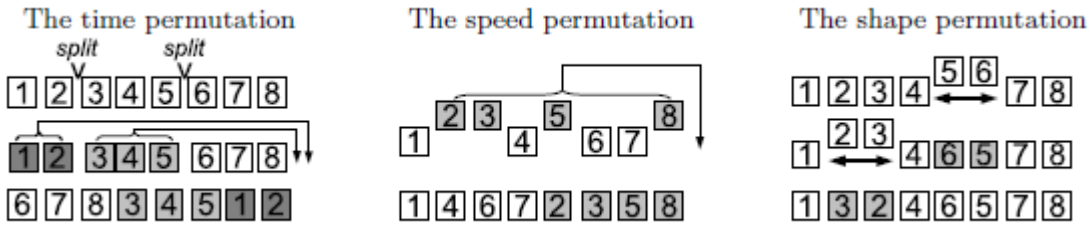

Figura 1.3.- Tipos de permutaciones controladas [49]

Este procedimiento se usa como complemento al prequential, para poder valorar con más eficacia la capacidad de adaptación a los cambios del sistema utilizado.

El desafío de este método se sitúa en realizar las permutaciones para alterar significativamente los datos, pero sin resultar completamente aleatorias para no arruinar la organización local de las instancias. Las métricas usadas para determinar la consecución de estos objetivos son la distancia total al vecino $D = \sum_{i=1}^{n-1} |j_i - j_{i+1}|$ y la distancia media al vecino $d = \frac{D}{n-1}$.

n denota la longitud de la cadena permutada y j_i el índice original de un elemento permutado que se ahora se encuentra en el índice i. Por lo tanto estas métricas analizan la separación original entre elementos que son actualmente contiguos.

En el trabajo se demuestra que las permutaciones definidas multiplican hasta por tres la distancia media, pero esta se mantiene independiente de la longitud de cadena n, mientras que una permutación aleatoria obtiene un incremento de la distancia media dependiente de n. Por lo tanto se cumple el objetivo de causar una perturbación significativa en los datos manteniendo la similitud con su estructura original.

1.5.3 Métricas para distribuciones especiales

En algunos casos la distribución de los datos posee alguna peculiaridad que disminuye la efectividad de las métricas tradicionales por lo que se hace necesario el uso de otras nuevas.

1.5.3.1 Distribución desigual de clases

En este caso la proporción de una o varias clases respecto de las demás es muy elevada. Si se tiene, por ejemplo, una clase que ocupa el 90% de las instancias, el resultado que se puede esperar es que incluso un clasificador rudimentario

pueda obtener una precisión muy elevada. Para aproximar mejor el rendimiento se utiliza la estadística Kappa:

$$\kappa = \frac{p - p_r}{1 - p_r}$$

$p = \frac{nº\ aciertos}{n}$, es la precisión del clasificador c con n el número de ejemplos. p_r es la precisión de un clasificador aleatorio y se define como [51]:

$$p_r = \sum_{i=1}^{n} P(y = i) P_c(\hat{y} = i)$$

$P(y = i)$ es la probabilidad de aparición de un ejemplo con clase i y $P_c(\hat{y} = i)$ la probabilidad de que el clasificador c etiquete el ejemplo con la clase i. El clasificador aleatorio asigna etiquetas a partir de la distribución del clasificador evaluado. Esto quiere decir que el clasificador aleatorio asigna el mismo número de etiquetas de cada clase que el original, pero puede variar los ejemplos a los que se les asigna.

En una distribución de clases muy desigual cabe esperar que el clasificador aleatorio, siguiendo la distribución del clasificador de referencia, alcance una precisión bastante elevada. El clasificador evaluado, por tanto, debería ofrecer una diferencia de precisión significativa para confirmar que su rendimiento no se debe al azar y poder valorar lo bien ajustado que está su modelo. El valor máximo de esta estadística es 1, que corresponde a un clasificador perfecto (con $p = 1$).

1.5.3.2 Datos con dependencia temporal

En general, los sistemas de clasificación aplicados a streams operan bajo la suposición de que la distribución de los datos es independiente, es decir, la probabilidad de recibir un dato de una clase concreta no tiene ninguna relación con la clase de los ejemplos anteriores. Esto se define formalmente como $P(y_t) = P(y_t \mid y_{t-1})$.

Esta suposición puede ser contraproducente a la hora de alcanzar un rendimiento óptimo. Tal y como se demuestra en [11], un clasificador sin cambios, que etiqueta los ejemplos con la clase del ejemplo anterior y cuya precisión se define como $p_{nc} = \sum_{i=1}^{n} P(y_t = i) P(y_t = i \mid y_{t-1} = i)$ [51], obtiene un rendimiento similar e incluso superior al de otros clasificadores reales.

Para tener en cuenta esta posibilidad, en [11] se define la métrica Kappa Plus, definida como:

$$\kappa = \frac{p - p_{nc}}{1 - p_{nc}}$$

La interpretación de esta métrica es similar a la de la estadística Kappa mencionada anteriormente. Se compara el rendimiento del clasificador con el de un clasificador sin cambios y la métrica determina hasta qué punto el clasificador analizado supera al aleatorio. Si el resultado de la métrica es negativo, significa que el clasificador sin cambios supera al de referencia, por lo que es muy probable que exista una dependencia temporal en los datos muy marcada, que debería ser explotada para incrementar la precisión.

1.5.4 Evaluación de la detección de la deriva conceptual

En [23] se presentan tres métricas para valorar la eficacia de la detección de la deriva conceptual:

- Posibilidad de detección de deriva: Se determina como el cociente entre el número de ocurrencias de deriva conceptual que se han detectado y las que se han producido realmente.
- Posibilidad de falsa alarma: Es el cociente entre el número de cambios detectados que efectivamente se estaban produciendo y el número de cambios detectados total.
- Retraso de detección: Mide la cantidad de tiempo (o de ejemplos procesados) que transcurre desde que se produce la deriva conceptual hasta que es detectada.

La aplicación de estas métricas, tal y como se señala en [43], tiene el problema de que requieren conocimiento previo de la distribución de los datos para poder determinarse de forma automática. Este conocimiento previo se logra usando conjuntos de datos creados de forma artificial, cuyas características pueden fijarse. En cualquier caso, es necesario usar a un experto humano que analice las diversas estadísticas generadas de forma gráfica para determinar las métricas anteriores.

1.6 Aplicaciones

A continuación se presentan una serie de áreas del conocimiento en las que se hace uso de algoritmos de aprendizaje automático adaptados a su uso en streams y se describe brevemente cuáles son sus características. Siguiendo la clasificación realizada en [50] se distinguen cuatro campos principales.

1.6.1 Monitorización y control

El requerimiento fundamental de esta clase de aplicaciones es la velocidad de procesamiento, ya que es necesario analizar el stream de datos en tiempo real para poder detectar con la mayor rapidez posible comportamientos extraños en sistema supervisado. En este entorno los datos están muy desequilibrados, ya que la proporción de comportamientos insólitos respecto a los normales es muy pequeña. Algunas de las aplicaciones específicas son:

Seguridad informática. La principal área de estudio es la detección de intrusiones en infraestructuras críticas. En este caso se analiza el tráfico de red para determinar acciones sospechosas. La fuente de deriva conceptual se halla en la aparición de nuevos tipos de ataque que explotan acciones que eran consideradas normales hasta el momento.

Finanzas. El objetivo aquí es la detección de fraude en el uso de tarjetas de crédito o actividad bancaria online. La detección se realiza mediante el análisis de las transacciones llevadas a cabo por un usuario para encontrar actividad extraña.

El problema se halla en que un usuario puede cambiar sus hábitos de forma natural sin que se haya producido fraude de ningún tipo. Para hacer frente a esto, se suele utilizar una clasificación ponderada respecto al coste de error. Esto significa que incluso si la probabilidad de que las acciones analizadas sean

normales es alta, se puede lanzar un aviso si el coste de ignorar el posible fraude es lo suficientemente elevado.

Industria. Esta clase de aplicaciones analizan datos de una red de sensores para determinar si un proceso o sistema está operando correctamente. El principal reto es lidiar con el comportamiento de los sensores: sus mediciones pueden ser incorrectas por lo que el algoritmo usado deberá ser robusto frente al ruido. Además un fallo en un sensor puede provocar una alarma aunque el funcionamiento sea normal.

1.6.2 Asistencia personal e información

Estos sistemas organizan y estructuran la información obtenida para aplicarla a varios campos como las recomendaciones personalizadas o la elaboración de perfiles de consumidores. En este ámbito no es necesario que el sistema opere en tiempo real y los costes de error son mucho más pequeños que en la monitorización.

Además las fronteras que separan la clasificación de instancias son mucho más difusas: las etiquetas asignadas no son absolutas sino que miden el grado de pertenencia de la instancia a la clase. Por ejemplo, un usuario puede ser aficionado a un cierto tipo de música y aun así no estar interesado en una recomendación de un artista del género.

Perfiles de consumidores. En este caso se acumula información de consumo o calificación de productos de diferentes usuarios para combinarlos, segmentar a los clientes y descubrir cuáles son más valiosos para el negocio, sus características demográficas y sus hábitos de compra. Este problema se basa fundamentalmente en el uso de técnicas de clustering. El cambio en las preferencias de los usuarios constituye la principal fuente de deriva conceptual.

Detección de spam. En el filtrado de spam la deriva conceptual no solo se debe al cambio en los intereses del usuario, sino también al cambio en las estrategias para su envío; las fuentes de spam modifican en contenido del mensaje, el asunto o el remitente para adaptarse a nuevos mecanismos de detección.

Otra cuestión a tener en cuenta es la presencia del cambio periódico en las preferencias del usuario. Por ejemplo, una persona puede no estar interesada en ofertas de viajes la mayor parte del año pero al llegar el verano esta clase de mensajes pueden ser de su agrado.

1.6.3 Soporte de decisiones

En este entorno la velocidad de procesamiento pasa a un segundo plano y se pone énfasis en la precisión de los modelos, ya que las aplicaciones trabajan en ámbitos en los que le coste de un error puede producir grandes pérdidas, como son la toma de decisiones estratégicas de grandes empresas.

Finanzas. Esta clase de aplicaciones realizan estimaciones de la solvencia de entidades económicas y su situación financiera a fin de determinar si es viable realizar una inversión. Es destacable que los enfoques mostrados en este campo no se suele tener en cuenta la presencia de deriva conceptual.

Biomédica. Existen varios campos de investigación como la predicción de resistencias emergentes en patógenos, la progresión de una enfermedad basándose en diversas constantes vitales o la autenticación biométrica. La deriva

conceptual en este caso proviene de la naturaleza de los microorganismos; pueden desarrollar resistencias al uso de fármacos y sufrir mutaciones que pueden variar su comportamiento.

1.6.4 Inteligencia artificial

En la inteligencia artificial el objetivo es lograr que los sistemas pueden interactuar con el entorno real y adaptarse a los cambios en este. Esta interacción directa con el medio físico obliga a que la velocidad de respuesta sea elevada y el error mínimo.

Domótica. Uno de los campos de aplicación es el desarrollo de las denominadas *smart homes*, cuyos componentes funcionan de forma autónoma. Estos sistemas deben adaptar su funcionamiento a las costumbres del usuario, de forma que reconozcan los patrones de uso y los implementen sin necesidad de recibir órdenes directas.

Realidad virtual. La creación de entidades inteligentes que interaccionan con el mundo virtual puede dotar de mayor autenticidad tanto a videojuegos como simuladores de diferentes tipos. Las diferentes conductas de los usuarios constituyen la principal fuente de deriva conceptual a la que tienen que enfrentarse los sistemas para adaptarse de forma dinámica a las diferentes interacciones.

2 Algoritmos de clasificación en Streams

A la hora de diseñar algoritmos para la clasificación sobre streams existen dos enfoques [7]:

- El método de envoltorio se centra en reutilizar al máximo los procedimientos preexistentes; en general apoyándose sobre clasificadores por lote tradicionales que son entrenados acumulando elementos del stream.
- El procedimiento de adaptación busca diseñar nuevos algoritmos a medida de las demandas impuestas por la naturaleza de los streams y tienen, en principio, un mejor rendimiento.

En el modelo de aprendizaje por lotes existen diferentes variedades de métodos como las redes neuronales, o árboles de clasificación. A continuación se presentan diferentes sistemas de clasificación en streams que se apoyan sobre dichos procedimientos y se examinan las técnicas usadas para satisfacer los requerimientos de esta clase de sistemas.

2.1 Árboles de decisión

Los árboles de decisión realizan una estructuración jerárquica de los datos que se crea recursivamente desde un nodo raíz hasta llegar a las hojas, que corresponden a las diferentes clases. Cada nodo del árbol contiene una comprobación del valor de un atributo concreto; de cada nodo surgen una cantidad de ramas igual a los diferentes valores del atributo.

El atributo seleccionado para realizar la partición es aquel que maximiza métricas como el coeficiente de Gini o la ganancia de información. La clasificación se lleva a cabo recorriendo el árbol desde la raíz, seleccionando en cada nodo la rama que corresponde al valor que tiene el elemento procesado en el atributo correspondiente, hasta llegar a una hoja que determina la clase que se le asigna.

El principal problema de su aplicación a streams es que los algoritmos tradicionales cargan el conjunto entero de datos en memoria. Esto en el caso de los streams es poco práctico debido a su enorme tamaño. Además también requieren evaluar todos los ejemplos a la hora de determinar qué atributo usar para la partición en cada nodo, incrementando el tiempo de ejecución y requiriendo varias pasadas sobre los datos.

2.1.1 VFDT

Una de las aplicaciones de árboles de decisión al contexto de los streams está en el VFDT(Very Fast Decision Tree)[20], que está basado en el del árbol de Hoeffding.

El árbol de Hoeffding se base en el uso de la desigualdad de Hoeffding. Esta desigualdad afirma que para una variable aleatoria r con un rango R sobre la que realizamos n observarciones y cuya media estimada es $\bar{r}$, con una probabilidad de $1 - \delta$, el valor real de la media es al menos $\bar{r} - \epsilon$ donde ϵ es:

$$\epsilon = \sqrt{\frac{R^2 \ln(\frac{1}{\delta})}{2n}}$$

δ es un parámetro fijado por el usuario que determina la confianza que se desea tener en la precisión de la estimación. La utilidad de esta desigualdad reside en que permite que usando n ejemplos se puede seleccionar, con la probabilidad de acierto deseada, el mismo atributo que se seleccionaría si se dispusiese del conjunto completo de datos. Concretamente, si $G(X_i)$ es la métrica, X_a el mejor y X_b el segundo mejor de los atributos candidato y $G(X_a) - G(X_b) > \epsilon$ entonces con probabilidad $1 - \delta$ se puede afirmar que el atributo considerado es el adecuado. Una vez se determina que se cumple a desigualdad anterior se crea una nueva partición en el nodo con el atributo X_a. El Algoritmo 2.1 detalla los pasos a seguir para generar el árbol.

Algoritmo 2.1.- VFDT [20]

VFDT

Entrada: S stream de ejemplos
X conjunto de atributos discretos que forman las entradas
G(.) función de evaluación
δ margen de error
Salida: HT árbol de decisión

1: HT = árbol con una sola hoja l_1 (la raíz)
2: X_1 = X ∪ X_0
3: $G_1(X_0) = G$ obtenida al predecir la clase más frecuente en S
4: **Para** toda clase y_k
5: **Para** cada valor x_{ij} de cada atributo $X_i \in X$
6: $n_{ijk}(l_1) = 0$
7: **Para** cada ejemplo (x, y_k) en S
8: Acomoda (x, y) en una hoja l usando HT
9: **Para** cada $x_{ij} \in x$ tal que $X_i \in X_l$
10: $n_{ijk}(l_1) = n_{ijk}(l_1) + 1$
11: Etiquetar la hoja l con la clase mayoritaria entre los ejemplos en l
12: **Si** los ejemplos en l no son todos de la misma clase
13: Calcular $G_l(X_i)$ para cada atributo $X_i \in X_l - \{X_0\}$ usando $n_{ijk}(l)$
14: Sea X_a y X_b el atributo con mayor y segundo mayor G_l
15: Calcular $\epsilon = \sqrt{\frac{R^2 \ln(\frac{1}{\delta})}{2n}}$
16: **Si** $G(X_a) - G(X_b) > \epsilon$ y $X_a \neq X_b$
17: Reemplazar l por un nodo que se particiona en X_a
18: **Para** cada rama de la partición
19: Añadir una hoja l_m y determinar $X_m = X - \{X_a\}$
20: $G_m(X_0) = G$ obtenida al predecir la clase más frecuente en l_m
21: **Para** toda clase y_k
22: **Para** cada valor x_{ij} de cada atributo $X_i \in X_m - \{X_0\}$
23: $n_{ijk}(l_m) = 0$
24: Devolver HT

El árbol mantiene una serie de estadísticas (denominadas $n_{ijk}(l)$) en cada hoja que son las que se usan para calcular la métrica G sin necesidad de tener

almacenados los ejemplos ni de revisarlos. Cuando se procesa una nueva instancia, se acomoda en una de las hojas basándose en el árbol creado hasta el momento y actualiza las estadísticas correspondientes a dicha hoja. Hecho esto se determina si la clase de todos los ejemplos de la hoja es la misma. Si no es así se considera que el árbol aún debe desarrollarse y se procede a calcular si se cumple la desigualdad $G(X_a) - G(X_b) > \epsilon$ sobre los atributos candidato. Si no es así, aún no se tiene la suficiente seguridad como para realizar una nueva partición y no se hace nada. Si se cumple entonces se crea una nueva ramificación sobre X_a y este se elimina de la lista de atributos candidato.

Con este sistema siempre se dispone de un árbol (aunque pueda no ser el definitivo) para poder realizar clasificación.

Además el VDFT realiza una serie de modificaciones para incrementar la eficiencia del árbol de Hoeffding:

- Durante la construcción del árbol la implementación original computa la métrica seleccionada con cada nuevo ejemplo. Para incrementar la velocidad de procesado se permite introducir un parámetro que exige recibir un número n_{min} de ejemplos antes de realizar la computación. Esto es útil puesto que es poco probable que se supere el umbral ϵ con la entrada de un solo ejemplo.
- El árbol solo realiza una partición cuando la diferencia entre los dos mejores atributos candidato es mayor que ϵ . En ciertos casos en los que las valoraciones están muy igualadas podría procesarse una gran cantidad de ejemplos antes de que se cumpliese la condición, lo que conlleva una ralentización en el desarrollo del árbol. Para solucionar el problema VFDT permite realizar la partición sobre el mejor atributo actual si $G(X_a) - g(X_b) < \epsilon < \tau$ siendo τ un umbral establecido por el usuario.
- Para incrementar el ahorro de memoria, los atributos menos prometedores, aquellos con una diferencia con el mejor mayor de ϵ, no se tienen en cuanta en los cálculos y sus estadísticas asociadas se descartan. La liberación de memoria también se aplica a los nodos menos prometedores.

2.1.2 CVFDT

El VFDT, sin embargo, no está pensado para lidiar con la deriva conceptual. Para solucionar este problema se creó el CVFDT[29]. Este algoritmo sigue las estrategias marcadas por VFDT pero añadiendo algunas modificaciones.

En primer lugar, las estadísticas para realizar el cálculo de la métrica se acumulan en cada nodo, no solo en las hojas. Este cambio se debe a que pueden crearse subárboles en cada nodo que pueden sustituir al original si ofrecen un mejor rendimiento.

Además el modelo de CVFDT permanece congruente con una ventana de ejemplos acumulados. Cuando la ventana está llena y se recibe un ejemplo, se descarta el más antiguo de la ventana. Este proceso se detalla en el Algoritmo 2.2.

Algoritmo 2.2.- Olvido de ejemplos en CVFDT [29]

OlvidarEjemplo

Entrada: HT árbol de decisión
(x_w, y_w) ejemplo a olvidar

1: Acomodar el ejemplo en usando HT
2: Sea P el conjunto de nodos atravesados
3: **Para** cada nodo $l \in P$
4: **Para** cada $x_{ij} \in x$ tal que $X_i \in X_l$
5: $n_{ijk}(l) = n_{ijk}(l) - 1$
6: **Para** cada árbol alternativo T_{alt} en l
7: OlvidarEjemplo(T_{alt}, (x_w, y_w))

Siguiendo el descarte se determinan los nodos por los que pasa el ejemplo en su clasificación, y, para cada uno de ellos, se actualizan las estadísticas almacenadas para considerar su ausencia. El resultado es que las decisiones tomadas en la construcción del árbol solo tienen en cuenta la información proporcionada por los ejemplos contenidos en la ventana.

La construcción de los subárboles se lleva a cabo según se detalla en el Algoritmo 2.3:

Algoritmo 2.3.- Verifcación de particiones en CVFDT [29]

VerificarValidez

Entrada: HT árbol de decisión
δ margen de error
τ coeficiente de resolución

1: **Para** cada nodo l en HT que no es una hoja
2: **Para** cada árbol alternativo T_{alt} en l
3: VerificarValidez(HT, δ)
4: Sea X_a el atributo de partición en l
5: Sea X_n y X_b el atributo con mayor y segundo mayor G_l al margen de X_a
6: Sea $\Delta G_l = G_l(X_n) - G_l(X_b)$
7: **Si** $\Delta G_l \geq 0$ y ningún T_{alt} de l tiene su partición raíz en X_n
8: Calcular ϵ como en la línea 15 del Algoritmo 2.1
9: **Si** $\Delta G_l > \epsilon$ o $\Delta G_l \geq \frac{\tau}{2}$ y $\epsilon < \tau$
10: Crear árbol con raíz en X_n como en las líneas 17-23 de VFDT

Este procedimiento se lleva a cabo de forma periódica y crea subárboles alternativos prometedores en el nodo analizado con raíz en X_n, que es el atributo más adecuado aparte del ya elegido. Las condiciones impuestas son muy similares a las de la determinar la partición de un hoja en VFDT pero más restrictivas, para evitar la creación de demasiados subárboles. Estos subárboles se desarrollan en el funcionamiento normal de algoritmo como se haría con el árbol principal, incluyendo el olvido de ejemplos cuando salen de la ventana. Esto incrementa el coste computacional pero permite crear un árbol más preciso

mediante la sustitución de partes del modelo para adaptarse a la deriva conceptual.

También de forma periódica se determina si es necesario sustituir un nodo por alguno de sus subárboles. Para cado nodo con un conjunto de subárboles alternativo no vacío se realiza un test con un número de ejemplos procedentes del stream. Si el subárbol más preciso supera al actual (con raíz en el nodo analizado) se lleva a cabo la sustitución. También se incluye un sistema de poda para aligerar la carga computacional y espacial del algoritmo: se eliminan los subárboles cuya precisión no se haya incrementado desde el último test, es decir, aquellos que no resultan prometedores.

Las decisiones de los atributos sobre los que se realiza la partición se basan en la estimación de la distribución de los datos en el instante, que puede cambiar debido a la deriva conceptual, por lo tanto es necesario disponer de un mecanismo para poder reajustar la estructura del árbol de modo que refleje fielmente las características de los datos en cada momento. Los subárboles proporcionan dicha funcionalidad, con la ventaja añadida de que cada uno de ellos se ha desarrollado en paralelo al árbol principal, por lo que no es necesario reconstruirlo desde cero.

2.2 Reglas de asociación

Las reglas de asociación son muy similares a los árboles de decisión [1], ya que los diferentes caminos que van desde la raíz hasta las hojas pueden expresarse en forma de reglas. La diferencia es que las reglas de asociación permiten que exista solapamiento entre las diferentes particiones, es decir, mientras que en los árboles cada ejemplo sigue un camino determinado según sus atributos existe la posibilidad de que más de una regla pueda ser aplicada a dicho ejemplo.

Tal y como se menciona en [18], los métodos apoyados en reglas de decisión tienen una ventaja en flexibilidad respecto a los árboles: en el caso de que ocurra deriva conceptual solo es necesario modificar o eliminar las reglas individuales afectadas, mientras que en los árboles se tendría que realizar una reconstrucción de la estructura. El resultado es un incremento en la eficiencia de computación.

2.2.1 LOCUST

El algoritmo LOCUST propuesto en [3] se aleja de los procedimientos habituales en la creación de reglas de asociación. El algoritmo acumula un bloque de datos procedente del stream y no los usa para elaborar un modelo mediante entrenamiento, sino que genera una serie de listas para cada atributo, divididas en secciones según el rango de valores.

Para cada atributo se determinan ϕ intervalos de igual amplitud sobre su dominio. Este dominio realmente es el intervalo acotado por el menor y el mayor valor para el atributo en el bloque de datos analizado. A medida que se incorporan nuevos bloques de datos los intervalos se modifican en consecuencia.

Cada atributo j tiene asociada una serie de listas V_i^j con $i \in \{1 \dots \phi - 1\}$ que determinan los intervalos en los que se ha dividido el dominio de dicho atributo. Cada lista V_i^j está divida en sublistas W_{ir}^j, que referencian a los ejemplos de la

clase r, con $r \in \{1 \dots k\}$, que entran dentro del intervalo definido por V_i^j. Este sistema, que evita la generación de un modelo global mediante la división del espacio de representación, está pensado para realizar la clasificación de forma específica para cada instancia, de acuerdo a los datos almacenados en cada momento. Esto se logra usando solo los datos similares a la instancia a clasificar, es decir, aquellos que pertenecen a las mismas listas V_i^j.

La creación de estas listas requiere dos pasadas sobre los datos: una para determinar la amplitud de los intervalos y otra para asociar los ejemplos a las listas generadas.

Tal y como se comentó en la sección 1.1, los algoritmos aplicados a streams solo deberían realizar una pasada sobre los datos, por lo que este procedimiento no es el óptimo. Los autores determinan que la solución al problema es crear las listas a partir de bloques de datos almacenados en memoria de un tamaño mínimo (5% de la memoria disponible).

La clasificación se realiza de forma iterativa realizando varios muestreos de conjuntos de atributos. El proceso se describe en el Algoritmo 2.4. Los atributos usados para cada muestra se seleccionan siguiendo un sesgo basado en el índice Gini específico de cada dimensión (atributo). Sea

$$f_r^j(T) = \frac{|W_{ir}^j|}{|V_i^j|}$$

la presencia fraccional de la clase r respecto a la dimensión j. T es la instancia a clasificar y V_i^j la lista a la que pertenece T. El índice de Gini específico a j se calcula como:

$$G^j(T) = \sum_{r=1}^{k} (f_r^j(T))^2$$

El índice Gini tiene valor máximo cuando la distribución de clases esté muy desequilibrada en el intervalo al que pertenece T. Este desequilibrio apunta a que existe un importante grado de similitud en el valor de la dimensión j en los elementos de la clase mayoritaria.

Para cada atributo seleccionado se realiza la intersección de los ejemplos contenidos en la lista V_i^j relevante, que es aquella a la que pertenece T, el ejemplo a clasificar. Al final del muestreo $U(Q)$ es un conjunto de ejemplos que se hallan dentro del mismo intervalo que T para cada una de las dimensiones seleccionadas. El proceso de selección de atributos en cada muestra continúa hasta que se alcanza una determinada confianza en que la clase dominante elegida es la adecuada o el número de ejemplos en $U(Q)$ disminuye por debajo del umbral establecido. La medida de confianza se define como:

$$\gamma(Q) = (f_d - f_g)\sqrt{\frac{f_g(1-f_g)}{|U(Q)|}}$$

Con f_d la presencia fraccional de la clase dominante en $|U(Q)|$ y f_g la presencia fraccional de dicha clase dominante sobre el total de los ejemplos.

Algoritmo 2.4.- LOCUST [3]

LOCUST

Entrada:
- T instancia de test
- n_s número de muestras
- t umbral de precisión
- n_t umbral de número de ejemplos

Para $i = 1$ hasta n_s
  $Q = \phi$
  Para todas las clases $l \in \{1 \dots k\}$
    $Count(l) = 0$
  Mientras $\gamma(Q) \leq t \ y \ |U(Q) > n_t|$
    Selecciona la dimensión j de forma aleatoria con sesgo
    Sea V_i^j la lista correspondiente a j que contiene T
    $U(Q) = U(Q) \cap V_i^j$
  Sea C_r la clase dominante en $U(Q)$
  $Count(r) = Count(r) + 1$
Etiqueta T con la clase con mayor valor de $Count(r)$

El conjunto de elementos fruto de las intersecciones es analizado para determinar cuál es la clase dominante, a la que se le añade un voto. El muestreo se repite el número de veces fijadas y la clase con más votos es la seleccionada para la clasificación. El resultado es que se crea, de forma implícita, una serie de reglas adaptadas a cada instancia de test, puesto que en cada iteración la lista de elementos final contiene los datos seleccionados mediante la discriminación de los valores de sus atributos, que serían los datos cubiertos por la correspondiente regla de decisión.

Para asegurar el lograr un equilibrio entre la precisión (mayor cantidad de iteraciones) y el tiempo de ejecución (menor número de iteraciones), LOCUST modifica el número de iteraciones del proceso de clasificación de forma adaptativa en función de la velocidad del stream. Para lograrlo almacena un histórico de los tiempos de ejecución de cada muestra del algoritmo y se determina t', el tiempo medio de procesamiento de una muestra para un ejemplo. Se desea procesar los ejemplos situados en una cola de tamaño q_c de modo que el tiempo necesario para procesar toda la cola sea igual a un tiempo de espera en la cola objetivo t_w, que se define como

$$t_w = q_c \, t' n_s$$

Con n_s el número de muestras por ejemplo. Por lo tanto el número de muestras se ajusta como $n_s = \frac{t_w}{q_c \, t'}$. Este valor se determina cada vez que cambia el tamaño de la cola.

Con el propósito de adaptarse a la deriva conceptual este algoritmo incorpora un sistema de olvido basado en un factor de envejecimiento exponencial, que elimina datos antiguos para poder basar el modelo solo en los más recientes.

La función usada es de la forma $2^{\lambda t}$, y determina la probabilidad de que uno de los bloques de datos almacenados en memoria sea usado para la clasificación. Concretamente, si han llegado t_c bloques, la probabilidad de que el bloque t se use es de $2^{\lambda(t-t_c)}$. En la práctica, esto se logra eliminando los bloques almacenados con probabilidad $2^{-\lambda}$ cada vez que se introduce uno nuevo.

2.3 Vecinos más cercanos

Esta clase de algoritmos se encuadran dentro del paradigma conocido como "lazy learning", cuya principal característica es que se descarta el proceso de entrenamiento. En estos métodos no se crea un modelo general a partir de los datos de entrenamiento, sino que se espera a disponer de los ejemplos de test para crear un modelo adaptado a cada instancia[1].

En concreto, el algoritmo de k vecinos más cercanos calcula las k instancias más cercanas a la entrada para una medida de distancia determinada y le asigna como clase aquella más común entre los vecinos calculados, por lo tanto, la asignación se basa en la similitud entre las instancias.

El procedimiento clásico requiere almacenar todos los ejemplos en memoria, así que los algoritmos adaptados a los streams aplican algún tipo de agregación de la información para limitar el espacio ocupado por los ejemplos; además dicha técnica también contribuye a la reducción de las operaciones requeridas, puesto que en el caso tradicional es necesario calcular la distancia del nuevo ejemplo respecto al resto.

2.3.1 On Demand Classification

Otro sistema usado es On Demand Classification[2], cuya principal característica es el uso del clasificador en diferentes horizontes de tiempo para obtener una clasificación óptima. La lógica de esta decisión se debe a que en un estado de cambio brusco usar únicamente los ejemplos más recientes resulta más efectivo que tener en cuenta el total de datos; la misma lógica se aplica a la situación inversa: en períodos de estabilidad aprovechar el máximo número de datos posible proporciona mejores resultados.

Este algoritmo utiliza y almacena microclústeres en vez de ejemplos individuales. Un microclúster M_p para un conjunto de ejemplos d-dimensionales $X_{i1}, \dots, X_{in}$, cuyos instantes de recepción son $T_{i1}, \dots, T_{in}$ es una tupla $(\overline{CF2^x}, \overline{CF1^x}, CF2^t, CF1^t, n, \ idclase)$ donde:

- $\overline{CF2^x}$ es un vector d-dimensional que contiene la suma de los cuadrados de los valores de los ejemplos que contiene M_p.
- $\overline{CF1^x}$ es un vector d-dimensional que contiene la suma de los valores de los ejemplos que contiene M_p.
- $CF2^t$ contiene la suma de los cuadrados de los instantes de tiempo de entrada de los ejemplos que contiene M_p.
- $CF1^t$ contiene la suma de los instantes de tiempo de entrada de los ejemplos que contiene M_p.
- n es el número de ejemplos que contiene M_p

- $idclase$ determina la clase del microclúster, que limita los ejemplos a los que puede contener.

Este enfoque de agregación colabora en la reducción de los requerimientos de memoria y computación y además es fácil de actualizar con la llegada de nuevos datos.

Mediante una ventana de tiempo geométrica se guardan los datos correspondientes al conjunto de microclústeres en diferentes instantes de tiempo; el objetivo es poder acceder a los modelos antiguos sobre diferentes horizontes temporales. Esta ventana funciona del siguiente modo:

Se dispone de una serie de marcos numerados $i \in \{0, \dots, \log_2 T\}$, con T la longitud máxima del stream. Como un stream es potencialmente infinito en lugar de T se define otra cifra que proporcione una capacidad adecuada a cada caso. Cada marco realiza una captura del estado de los microclústeres si $t \ mod \ 2^i = 0$ y $t \ mod \ 2^{i+1} \neq 0$, siendo t el instante de tiempo. Los marcos tienen una capacidad máxima definida por el usuario, así que si esta se supera, se elimina la captura más antigua del marco. Se muestra un ejemplo en la Figura 2.1.

Frame no.	Snapshots (by clock time)
0	69 67 65
1	70 66 62
2	68 60 52
3	56 40 24
4	48 16
5	64 32

Figura 2.1.- Ventana de tiempo geométrica **[2]**

Una pequeña parte (1%) de los datos recibidos es reservada para realizar tests de precisión que determinen el horizonte temporal óptimo para la clasificación. Este conjunto se actualiza para incorporar los datos más recientes.

Un conjunto de datos es acumulado para realizar la inicialización de un número igual de microclústers para cada clase. El proceso de actualizar el modelo, que se esquematiza en el Algoritmo 2.5, es simple. Para cada dato de entrada se calcula el clúster más cercano de su clase, y se determina que esa distancia está dentro de un umbral. Este umbral se define como la desviación cuadrática del conjunto t de puntos pertenecientes al cluster respecto de su centroide. Esta medida permite determinar si el nuevo punto se encuentra dentro de lo que se considera una distancia normal al centroide del clúster o si realmente no puede considerarse parte de este.

Si el ejemplo está a una distancia aceptable del clúster más cercano se añade a este y se actualiza su vector de datos. Si la distancia del punto al clúster supera el umbral se crea un nuevo clúster que contiene al punto. Como el número de clústeres debe ser constante, si se crea uno nuevo es necesario borrar o fusionar otros. La definición de un número fijo de clústeres evita que se dispare el espacio de almacenamiento en el caso de que los datos sean muy dispares y se creasen demasiados.

Algoritmo 2.5.- Actualización de On Demand Classification

On Demand		
Entrada:	S	Stream de ejemplos
	B	Bloque de datos inicial
	q	número de microclusters máximo

1: Inicializar los microclusters con k-means a partir de B
2: **Para** cada ejemplo $x \in S$
3: Calcular la distancia de x los microclusters de su misma clase
4: **Si** todas las distancias son mayores que un umbral
5: Crear un nuevo microcluster a partir de x
6: Determinar la relevancia temporal de todos los microclusters
7: **Si** la menor relevancia es menor que un umbral
8: Eliminar el microcluster correspondiente
9: **Si no**
10: Hallar la distancia entre los microclusters de igual clase
11: Combinar aquellos dos cuya distancia sea mínima
12: **Si no**
13: Actualizar el vector de datos del clúster más cercano a x

La eliminación de clústeres se realiza teniendo en cuenta su relevancia para el modelo actual, que se mide a partir de los instantes de tiempo de los últimos datos añadidos al clúster. Un clúster que no recibe incorporaciones no se ajusta a la distribución actual de los datos, que puede variar debido a la deriva conceptual, y por tanto no resulta útil.

La media de los instantes de tiempo de las últimas m incorporaciones se determina utilizando la media de los instantes de llegada globales $\mu = CF1^t/n$ y la desviación típica $\sigma = \sqrt{\frac{CF2^t}{n} - \mu^2}$. La relevancia temporal se calcula a partir del $m/2n$ -ésimo percentil de los datos de cada cluster. Se elimina el cluster con la menor relevancia temporal que no supera un umbral definido por el usuario.

Si ningún cluster falla la prueba anterior (están por encima del umbral) se realiza una fusión de dos de ellos de la misma clase cuya distancia entre sí sea la mínima.

A la hora de realizar clasificaciones es cuando este algoritmo calcula cual es el horizonte temporal óptimo. Para ello se usa una propiedad de los clústeres denominada propiedad substractiva. Formalmente:

Si tenemos dos conjuntos de puntos C_1 y C_2 tales que $C_1 \supseteq C_2$, entonces el vector de datos del cluster que contiene los puntos $C_1 - C_2$ se define como $\overline{CFT(C_1 - C_2)} = \overline{CFT(C_1)} - \overline{CFT(C_2)}$

Esto se traduce en que es posible obtener las características de un clúster que contiene solamente los ejemplos recibidos en un intervalo temporal concreto.

El instante de tiempo actual se denota como t_c y cada uno de los instantes de tiempo de las capturas realizadas en la ventana de tiempo geométrica se denominan t_h; el algoritmo determina el vector de datos de los microclusters

para los horizontes temporales $(t_c - t_h, t_c)$. El rendimiento de cada uno de ellos se determina mediante un test usando el conjunto de datos reservado para este propósito. Los p (valor determinado por el usuario) horizontes temporales que ofrezcan mayor precisión serán usados para realizar la clasificación de nuevos ejemplos mediante un sistema de votación.

Este enfoque no solo se adapta a la deriva conceptual tanto brusca como gradual, sino que también ofrece robustez en el caso de que la distribución de las clases en el stream fluctúe en el tiempo: sin no han llegado ejemplos de una clase últimamente y estos reaparecen, usar un horizonte de tiempo que incluya instancias de dicha clase mejorará la precisión.

2.4 SVM

Las SVM son un tipo de clasificador que basa su funcionamiento en la separación de las instancias de las clases mediante condiciones lineales.

Su funcionamiento es el siguiente: se consideran los ejemplos como puntos en un espacio n-dimensional, donde n es el número de atributos de dichos ejemplos. La SVM determina un hiperplano que separa las instancias de las diferentes clases en el espacio de representación (generalmente se usan para clasificación binaria). Dicho hiperplano debe maximizar la distancia respecto a los puntos más cercanos, denominados vectores de soporte, por lo que el separador se obtiene resolviendo un problema de optimización.

En el caso de que los ejemplos no sean linealmente separables se procede a realizar una transformación de los puntos a un espacio de dimensión superior, en el que si se podrán separar. Para realizar dicha transformación se recurre al uso de las denominadas funciones kernel, cuya ventaja reside en que permiten calcular el producto de dos vectores en un espacio de mayor dimensión sin realizar ningún transformación de forma directa.

Uno de los problemas a los que se enfrenta la aplicación de SVM al ámbito de los streams es que el cálculo del separador es costoso; el problema de optimización requiere realizar cálculos para todos los ejemplos lo que deja la complejidad temporal en $O(N^3)$ y la espacial en $O(N^2)$ [39]. Para un entorno que requiere una velocidad de respuesta elevada y cuyo volumen de datos es potencialmente infinito el coste no es aceptable. Estas características hacen que su adaptación a streams sea más costosa que la de otros métodos por lo que el estudio en este campo no está tan desarrollado como el de los ensembles o árboles de decisión.

2.4.1 Adaptación al modelo incremental

Una de las propuestas para reducir el tiempo de computación y almacenamiento es aplicar el modelo de aprendizaje incremental para actualizar el modelo con un menor número de ejemplos a la vez. El trabajo realizado en [19], [45] implementa el sistema tomando bloques de datos, realizando el entrenamiento y almacenando los vectores de soporte calculados, que se incluirán juntos a los datos del siguiente bloque de entrenamiento. Solo se conservan los vectores de soporte porque son los únicos puntos que resultan relevantes a la hora de definir el hiperplano.

[19] también ofrece una breve descripción de la aplicación del modelo incremental propuesto al caso de los streams. El sistema funciona agrupando las

entradas en bloques de tamaño fijo y manteniendo un número fijo de SVM. Por cada nuevo bloque entrante se elimina la SVM más antigua, se actualizan las demás y se crea una nueva a partir de dicho bloque; así se logra mantener los clasificadores consistentes con la distribución actual de los datos para adaptarse a la deriva conceptual.

2.4.2 Blurred Ball SVM

Otro enfoque para adaptar las SVM al ámbito de los streams consiste en transformar el problema de optimización en el de la mínima circunferencia de recubrimiento[36]. Este problema se basa en encontrar una circunferencia del mínimo radio que recubra una serie de puntos. La transformación se basa en que si la función kernel $K(x,y) = \varphi(x)\varphi(y)$ usada cumple que $K(x,x) = k$, donde k es una constante, se puede hallar el hiperplano óptimo w hallando la esfera mínima de recubrimiento de los puntos a los que se les aplica la transformación $\bar{\varphi}(x_i, y_i)$ definida del siguiente modo:

$$\bar{\varphi}(x_i, y_i) = \begin{bmatrix} y_i\varphi(x_i) \\ y_i \\ 1 \\ \frac{1}{\sqrt{C}} e_i \end{bmatrix}$$

e_i es el vector i de la base canónica (ceros en todas las posiciones excepto en la i, que contiene un uno) y C es un parámetro determinado por el usuario que determina la tolerancia a errores en clasificación. Si c es el centro óptimo de la esfera, entonces $w = c.(e_1, \dots, e_m)$.

El principal atractivo de este enfoque es la reducción en la complejidad espacial, que pasa a ser independiente del número de instancias: sólo es necesario mantener en memoria el conjunto de puntos que definen la circunferencia de recubrimiento, denominado conjunto central. Este conjunto contiene los puntos tales que la circunferencia que recubre dichos puntos es lo suficientemente aproximada a la circunferencia que recubre al total de ejemplos. También se reduce el coste computacional de forma notoria debido a que el problema de optimización se reduce a calcular si los puntos están contenidos dentro de una circunferencia.

Cabe destacar que este algoritmo es no supervisado, es decir, no tiene en cuenta la clase que tiene cada instancia a la hora de procesarla.

El Algoritmo 2.6 detalla el funcionamiento de este sistema. Se trabaja manteniendo un conjunto de circunferencias definidos por sus conjuntos centrales. Los ejemplos se transforman por el procedimiento descrito anteriormente y se acumulan en un buffer hasta que este se llena, que es cuando se actualiza el modelo. La lógica del uso del buffer está en el ahorro de tiempo de computación; actualizar el modelo ejemplo a ejemplo podría obligar a calcular una nueva circunferencia en cada iteración en el peor de los casos.

Una vez el buffer alcanza la capacidad definida se determina si alguno de los ejemplos está fuera de la $1 + \varepsilon$ expansión de todas las circunferencias almacenadas, es decir, si alguno de los puntos se halla a una distancia mayor de $(1 + \varepsilon)R$ del centro de cada circunferencia, donde R es su radio.

En este caso se combinan los puntos almacenados en el buffer con los pertenecientes a los conjuntos centrales existentes y se determina el conjunto central de una nueva circunferencia que los recubra a todos. Hecho esto se procede a descartar todos los conjuntos centrales cuya circunferencia asociada tenga un radio que no supera al radio de la nueva esfera en un umbral. Se considera las circunferencias eliminadas tienen un radio demasiado pequeño para poder cubrir a ningún punto nuevo y, por tanto, ya no resultan relevantes. El buffer se vacía y el proceso se repite.

Algoritmo 2.6.- Blurred Ball SVM [36]

Blurred Ball SVM

Entrada: (x_i, y_i) ejemplo
L longitud de la ventana de ejemplos
ε margen

1: centrales = []
2: buffer = ∅
3: calcular $x' = \bar{\varphi}(x_i, y_i)$
4: Añadir x' a L
5: **Si** la longitud de buffer es menor que L
6: Acabar
7: **Si** $\exists x' \in buffer$ que no está en la $1 + \varepsilon$ expansión de ninguna circunferencia en centrales
8: c = nuevo conjunto central sobre *buffer* ∪ *centrales* que define a B
9: Descartar todos los conjuntos centrales cuya circunferencia tenga un radio menor que $r(B) * \varepsilon/4$
10: Centrales = centrales ∪ c
11: buffer = ∅

El algoritmo produce varios conjuntos centrales, cada uno de ellos equivalente a un hiperplano. Para realizar la clasificación sus predicciones se combinan del siguiente modo:

El apoyo de un punto p se define como $Sup(p) = \{B \in centrales | \, p \in B\}$, es decir, el apoyo contiene a los conjuntos centrales cuya circunferencia asociada contiene a p.

La puntuación de un punto p de define como:

$$S(p) = \sum_{B \in Sup(p)} p.\frac{c_B}{\|c_B\|}$$

Siendo c_B el centro de la circunferencia B. Por tanto $S(p)$ es la suma de las distancias de p a los separadores definidos por B.

Considerando que las clases son +1 y -1 (problema de clasificación binaria), la predicción es: $sgn(S(p) - S(-p))$.

2.5 Redes Neuronales

Las redes neuronales son un tipo de constructo inspirado en el comportamiento de las neuronas biológicas con el objetivo de resolver problemas en el dominio del aprendizaje automático. La unidad básica de la que están compuestas es la neurona, que tiene asociada una serie de pesos asociados a cada entrada. Cada neurona calcula la suma ponderada de sus entradas y le aplica una función, como por ejemplo la sigmoide, cuyo resultado es la salida de dicha neurona. En general las redes neuronales tienen una capa de nodos de entrada, que no realizan ninguna operación más que la transmisión de los datos; una serie de capas internas, cuyas neuronas operan del modo descrito anteriormente; y una capa de salida, cuyos nodos también realizan cálculos y cuyos resultados se usan para determinar la solución del problema.

El proceso de aprendizaje de las redes neuronales se basa en la actualización de los pesos en las entradas de las neuronas (que suelen inicializarse de forma aleatoria) basándose en los errores cometidos a la hora de clasificar muestras. Este procedimiento es bastante costoso computacionalmente e implica tener que realizar varias pasadas sobre los datos hasta que se realice una clasificación adecuada o supere un cierto número de ejecuciones, por lo tanto resulta poco adecuado ya que el entorno de streams requiere de una sola pasada sobre los datos y un tiempo de ejecución limitado. Sin embargo, tal y como se señala en [16] este procedimiento no es necesario en el caso de los streams, donde la abundancia de ejemplos permite realizar un ajuste preciso realizando una única pasada sobre los datos. No obstante el problema de la complejidad temporal permanece, por lo que generalmente se usan redes neuronales con un número limitado de capas internas.

Además las redes neuronales cuentan con la ventaja de unos requerimientos de memoria modestos debido a que el sistema de entrenamiento descrito es incremental, de modo que no será necesario almacenar ejemplos en memoria. Dicho modelo incremental hace que la adaptación de redes neuronales al entorno de los streams sea bastante directa[16].

2.5.1 DELM

El trabajo en [47] propone el uso de una Extreme Learning Machine(ELM) de dos capas internas adaptada al uso en streams.

Una ELM es un tipo de red neuronal en la cual los pesos de las conexiones entre la capa de entrada y las capas internas son inicializados aleatoriamente y no se modifican durante su uso. Los pesos de las conexiones con la capa de salida se calculan de forma directa. Este enfoque tiene la ventaja de ofrecer una mayor rapidez en la actualización del modelo al no tener que hacer uso de algoritmos como backpropagation para ajustar los pesos de las conexiones con cada pocos ejemplos. Por esta misma razón las ELM tienen, en general, mejores capacidades de generalización que otros tipos de redes[26].

Teniendo un bloque de ejemplos $D = \{(x_1, y_1), \dots, (x_n, y_n)\}$, siendo x_i un vector de atributos de longitud N; y_i una de las m clases y L el número de nodos de la capa interna de la red; la salida se puede calcular como:

$$f(x_i) = \sum_{j=1}^{L} \beta_j g(a_j, b_j, x_i) = y_i \; i = 1, \dots, N$$

$a_j = (a_{ji}, \dots, a_{jn})$ es el vector de pesos de las conexiones de los nodos de entrada con el nodo interno j; b_j es el sesgo del nodo j y β_j el vector de pesos de las conexiones del nodo interno j con los nodos de la capa de salida.

Como DELM trabaja con bloques de datos resulta más sencillo expresar la ecuación anterior en forma matricial:

$$H\beta = T$$

$$H = \begin{bmatrix} g(a_1, b_1, x_1) & \dots & g(a_L, b_L, x_1) \\ \vdots & \ddots & \vdots \\ g(a_1, b_1, x_N) & \dots & g(a_L, b_L, x_N) \end{bmatrix}; \beta = \begin{bmatrix} \beta_{11} & \dots & \beta_{1m} \\ \vdots & \ddots & \vdots \\ \beta_{L1} & \dots & \beta_{Lm} \end{bmatrix}; T = \begin{bmatrix} T_{11} & \dots & T_{1m} \\ \vdots & \ddots & \vdots \\ T_{N1} & \dots & T_{Nm} \end{bmatrix}$$

H es la salida de la capa interna de la red; β contiene los pesos de las conexiones de los nodos de la red interna con cada uno de los m nodos de salida y T contiene las etiquetas de los ejemplos codificadas como vectores que tienen ceros en todas las posiciones salvo en la correspondiente a la clase, que tiene un uno.

β se define como $\beta = H^{\dagger}T$; con $H^{\dagger}$ la pseudoinversa de Moore-Penrose. β se obtiene como la solución de un problema de optimización definido del siguiente modo:

$$\min \frac{1}{2}\|\beta\|^2 + \frac{C}{2}\sum_{i=1}^{N}\|\varepsilon_i\|^2$$

$$\text{Sujeto a: } H(x_i)\beta = T(x_i) - \varepsilon_i$$

C es un factor de penalización definido por el usuario y ε_i el error de clasificación; definido como un vector que contiene la diferencia entre los valores obtenidos y esperados (básicamente la matriz T) de los nodos de la capa de salida. El problema de optimización no incluye solo el error, sino que también se intenta minimizar la magnitud de los pesos de la capa de salida a fin de lograr una mejor capacidad de generalización del modelo. La relevancia del parámetro C es que determina la tolerancia a los errores de clasificación: un valor pequeño disminuirá su impacto en la minimización.

La solución al problema es la siguiente:

$$\begin{cases} \beta = H^T(\frac{I}{C} + HH^T)^{-1}T \;\; si \; L \geq N \\ \beta = (\frac{I}{C} + H^T H)^{-1}H^T T \;\; si \; L < N \end{cases}$$

I es la matriz identidad de las dimensiones adecuadas. La inclusión del término $\frac{I}{C}$ se realiza para obtener una mayor capacidad de generalización en el modelo creado, tal y como se explica en [26].

La matriz H es $N \; x \; L$ por lo que HH^T es $N \; x \; N$ y $H^T H$ es $L \; x \; L$. El criterio de selección entre los dos métodos de cálculo se determina para minimizar la

dimensionalidad de la matriz resultante y así ahorrar tiempo de cálculo, ya que la multiplicación de matrices es una operación computacionalmente compleja.

La estructura de la DELM se muestra en la Figura 2.2.

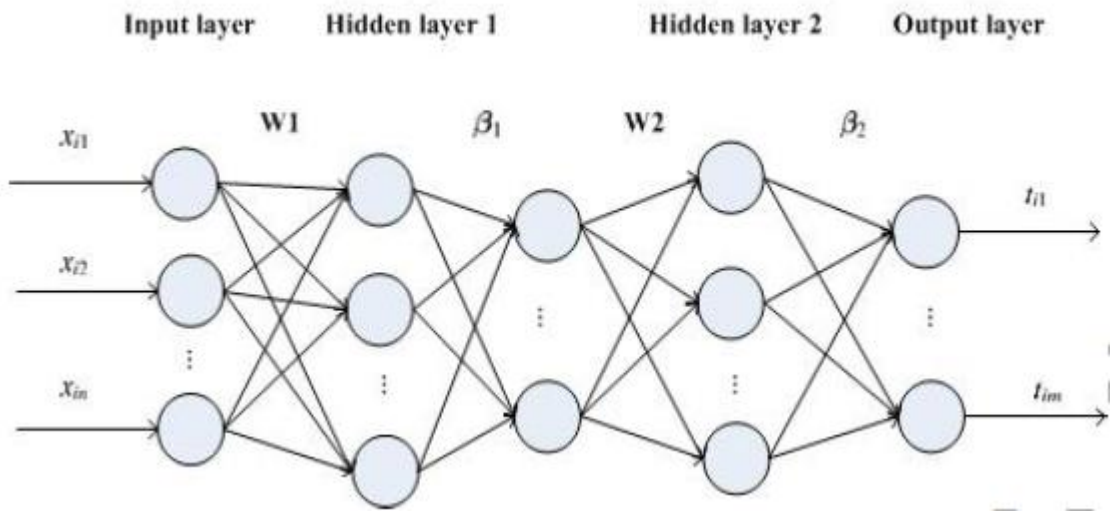

Figura 2.2.- Estructura de DELM [47]

En el Algoritmo 2.7 se explican los pasos de funcionamiento. Se opera acumulando un bloque de datos en una ventana de un tamaño determinado. El primer bloque recibido se usa para crear el modelo. Como hay dos capas internas existirán dos matrices H y β, una para cada capa. β_1 denota los pesos de las conexiones de la primera capa interna con la segunda y β_2 los de la segunda capa interna con la de la salida. Tanto β_1 como β_2 se calculan siguiendo el método establecido. Las matrices H_1 y H_2 son ambas obtenidas de forma aleatoria, como exige la definición de una ELM.

Para los siguientes bloques de datos se realizará el aprendizaje del modelo de forma prequential; primero realizando un test para comprobar la precisión y luego actualizando con esos mismos datos de test. Se almacena la probabilidad de error p_i y su desviación típica $s_i = \sqrt{\frac{p_i(1-p_i)}{i}}$

La actualización de β_1 y β_2 se realiza del siguiente modo:

$$\beta'_1 = \beta_1 + (K')^{-1} H_1^T (T - H_1\beta_1)$$

$$K' = K + H_1^T H_1 \text{ con valor inicial } K_0 = \frac{I}{C} + H_1^T H_1$$

$$\beta'_2 = \beta_2 + (Q')^{-1} H_2^T (T - H_2\beta_2)$$

$$Q' = Q + H_2^T H_2 \text{ con valor inicial } Q_0 = \frac{I}{C} + H_2^T H_2$$

DELM se adapta fácilmente a la deriva conceptual gradual debido a que su modelo se actualiza constantemente, pero es necesario tener un mecanismo que detecte cambios más bruscos, así que se incluye un sistema de detección de la deriva conceptual similar al DDM descrito en la sección 1.4.2, con la diferencia de que en este caso se añade un umbral τ para limitar no solo la variación del error sino también su magnitud. Cuando se detecta que puede estar

produciéndose deriva conceptual (estado de aviso) el sistema incrementa el número de nodos en las capas internas para intentar adaptarse a los cambios y cuando se ha confirmado la presencia de deriva se reentrena el modelo para que se ajuste a la nueva distribución.

Algoritmo 2.7.- DELM [47]

DELM		
Entrada:	S	stream de datos
	w	tamaño de la ventana
	τ	umbral de error

1: Entrenar la red con un bloque de datos
2: **Mientras** S no sea vacío
3: Almacenar un bloque B de tamaño w
4: Testear el modelo con B
5: Calcular p_i y s_i
6: Actualizar p_{min} y s_{min}
7: **Si** $p_i + s_i < p_{min} + 2s_{min}$ y $p_i < \tau$
8: Actualizar β_1 y β_2
9: **Si no si** $p_i + s_i \geq p_{min} + 2s_{min}$ y $p_i + s_i < p_{min} + 3s_{min}$
10: Actualizar β_1 y β_2
11: Actualizar el número de nodos de las capas internas
12: **Si no si** $p_i + s_i \geq p_{min} + 3s_{min}$ o $p_i < \tau$
13: Actualizar el número de nodos de las capas internas
14: Descartar el modelo actual y reentrenarlo con B

La actualización del número de nodos de las capas internas se realiza de forma diferente según el estado. En el caso de que $p_i + s_i \geq p_{min} + 2s_{min}$ y $p_i + s_i < p_{min} + 3s_{min}$ (estado de aviso) el número de nodos de la primera y segunda capas internas, N_1 y N_2 respectivamente, se calculan como:

$$N_2 = N_1 \text{ y } N_1 = \min(w, N_1 + floor(\frac{c}{p_i + \varepsilon}))$$

Si $p_i + s_i \geq p_{min} + 3s_{min}$ o $p_i < \tau$ (deriva detectada):

$$N_1 = \min(w, N_1 + floor(\frac{c}{p_i + \varepsilon})) \text{ y } N_2 = \min(w, N_2 + floor(\frac{c}{p_i + \varepsilon}))$$

c es una constante establecida por el usuario y ε una pequeña constante para evitar la división entre cero.

En el caso de que se detecte deriva conceptual se considera que el modelo se encuentra demasiado desfasado y se elabora otro nuevo con el bloque de datos actual.

El cálculo del número de neuronas mediante un mínimo sirve para mantener acotado el tiempo de ejecución y el consumo de memoria en caso de que se detectase deriva de forma continua y se incrementase descontroladamente dicho número. De este modo, el número de neuronas en cada capa es como máximo igual al tamaño de ventana seleccionado.

2.6 Ensembles

Los ensembles son un conjunto de clasificadores (también denominados expertos) que operan de forma conjunta para tomar decisiones. Generalmente los resultados individuales se combinan mediante un procedimiento de votación.

La lógica del uso de ensembles reside en las ventajas que aportan en la clasificación. Tal y como se describe en [27], los ensembles ofrecen una precisión mayor que un clasificador único cuando los miembros del ensemble individuales tienen una precisión mayor que la de un clasificador aleatorio (probabilidad de error igual a 0.5) y no existe correlación entre los errores de los clasificadores individuales. Según [21] los errores de dos modelos están correlacionados cuando ambos etiquetan un ejemplo de la clase y_i como perteneciente a una clase distinta y_j.

Además los ensembles también permiten una mejor adaptación a la deriva conceptual. Son capaces de incluir información actualizada del stream mediante la inclusión de un nuevo miembro mientras el conocimiento más antiguo se conserva en el resto. Un clasificador individual estaría más limitado en este aspecto puesto que tiene un único modelo.

Las desventajas que se imponen son evidentes: un incremento en el espacio de almacenamiento y tiempo de procesamiento. A causa de lo anterior el uso de ensembles se limita a entornos en los que no se requiere el procesamiento de datos en tiempo real pero se requiere tanta precisión como sea posible.

2.6.1 AUE

Otro ejemplo se halla en el algoritmo AUE (Accuracy Updated Ensemble)[14], que es una actualización del AWE (Accuracy Weighted Ensemble)[46].

La idea fundamental de ambos es la determinación del conjunto de datos que debería ser usado para lograr una clasificación óptima. Los sistemas basados en ventanas eliminan ejemplos basándose en su antigüedad, sin embargo, tal enfoque puede no ser adecuado en ciertos casos, por lo que los autores defienden que la conservación de los ejemplos debe basarse en su relevancia para la distribución actual de datos y no solo su antigüedad. Para lograr esta meta los autores proponen el uso de un ensemble de clasificadores. Estos clasificadores están entrenados sobre diferentes bloques de datos, por lo que los pesos asignados a cada uno de ellos determinarán la importancia de los diferentes datos en la clasificación presente.

AUE opera sobre bloques de datos. Para cada nuevo bloque, el algoritmo crea un nuevo clasificador candidato y comprueba su precisión y la del resto de expertos. La creación de candidatos a partir de los datos más recientes constituye un enfoque más activo y persistente para la adaptación a la deriva conceptual que simplemente esperar a que el error se dispare para realizar adiciones al ensemble, como ocurre en otros algoritmos.

El error se define como el error cuadrático medio:

$$MSE_i = \frac{1}{|B_i|} \sum_{(x,y)\in B_i} (1 - f_y^i(x))^2$$

B_i es el bloque de datos sobre el que se evalua y $f_y^i(x)$ es la probabilidad de que el clasificador C_i asigne la clase y al ejemplo x.

Como se comentó anteriormente, la idea fundamental es usar un modelo basado en datos relevantes para la distribución actual. Tal y como se demuestra formalmente en [46], esto se consigue calculando los pesos de los expertos para cada nuevo bloque de datos de forma inversamente proporcional a su error:

$$w_i = \frac{1}{MSE_i + \epsilon}$$

ϵ es una pequeña constante para evitar la división por cero. La asignación de peso basado en la precisión garantiza que el ensemble producirá menos errores que un clasificador individual. El Algoritmo 2.8 describe el funcionamiento.

Algoritmo 2.8.- AUE [14]

DWM

Entrada: S Stream de ejemplos
k número de clasificadores

1: $C = \emptyset$ //Conjunto de clasificadores
2: **Para** todos los bloques $B_i \in S$
3: Crear clasificador C' a partir de
4: Cálculo del error MSE de C' sobre B_i
5: Calcular el peso w' de C'
6: **Para** todos los clasificadores $C_i \in C$
7: Calcular MSE_i sobre B_i
8: Calcular w_i
9: $\varepsilon = k$ clasificadores con mayor peso en $C \cup \{C'\}$
10: $C = C \cup \{C'\}$
11: **Para** todos los clasificadores $C_e \in \varepsilon$
12: **Si** $w_e > \frac{1}{MSE_r}$ y $C_e \neq C'$
13: Actualizar C_e con B_i

Los k mejores clasificadores son los que forman parte del ensemble. Al contrario que AWE, AUE integra un proceso de actualización de los clasificadores como parte del algoritmo. La actualización solo se aplica a los clasificadores del ensemble con un error menor que el de un clasificador aleatorio que realiza sus predicciones basándose únicamente en la distribución de los datos:

$$MSE_r = \sum_y p(y)(1 - p(y))^2$$

$p(y)$ es la probabilidad de aparición de un ejemplo de la clase y.

El propósito de esta restricción en la actualización es doble: por un lado se desea aumentar la precisión de los clasificadores mejor adaptados a la distribución actual de los datos proporcionándoles más ejemplos; por otro, se quiere mantener el modelo de los demás clasificadores, no tan bien adaptado a los ejemplos actuales, en caso de que pueda ser útil en el futuro.

Esta posibilidad de actualización elimina uno de los problemas de AWE, que era la selección de un tamaño de bloque. Al no soportar la actualización de los clasificadores existentes, era necesario determinar un tamaño de bloque suficientemente grande para que las precisiones superasen un nivel aceptable pero no lo suficientemente grande como para que la detección de deriva conceptual pasase desapercibida. La única opción era determinar el tamaño óptimo realizando varias pruebas.

3 Implementación y evaluación

Para obtener una idea aproximada del comportamiento de los algoritmos aplicados a streams se va a programar uno de ellos y evaluar su rendimiento sobre diferentes sets de datos y con diferentes parámetros. Además se procederá a compararlo con un equivalente tradicional. El algoritmo elegido es una ELM al estilo del discutido en el epígrafe 2.5.1. Para realizar la comparación se usa uno de los modelos de redes más usados y el más similar al seleccionado: el perceptrón multicapa (denominado MLP por sus siglas en inglés). Más información sobre la implementación realizada se puede encontrar en el apéndice A.

3.1 Software utilizado

El lenguaje escogido para realizar el trabajo es Python[41], que es un lenguaje libre, ofrece soporte para muchos paradigmas de programación como la orientación a objetos o la programación funcional y dispone de una gran cantidad de librerías para diversos propósitos, entre ellas algunas dedicadas al aprendizaje automático. Se ha utilizado la versión 3.6 del lenguaje.

La principal herramienta usada para realizar la implementación de la ELM es Numpy[37], una librería de cálculo científico que permite trabajar de forma fácil con arrays multidimensionales. Además incluye soporte para su integración con diferentes Subprogramas de Álgebra Lineal Básica (BLAS) para optimizar el rendimiento de las operaciones. Se ha utilizado la versión 1.12 enlazada con la Intel Math Kernel Library.

Para la programación del perceptrón multicapa se ha usado Keras[17], una API de alto nivel para el trabajo con redes neuronales. Keras permite al programador definir y parametrizar una red neuronal en unas pocas líneas de código. La versión usada es la 2.0.4.

Para el ajuste de los sets de datos usados y su lectura se ha empleado pandas[35], una librería de análisis y manipulación de datos. Provee estructuras de datos sencillas de usar y optimizadas gracias a la integración con C. La versión usada es la 0.19.2.

Las gráficas han sido generadas con la librería matplotlib[30], que permite realizar multitud de representaciones visuales de los datos basándose en una interfaz similar a la de MATLAB. Se ha usado la versión 2.0.0.

Por último la herramienta psutil (5.2.2)[40] se ha aprovechado para perfilar el consumo de memoria de los algoritmos usados.

3.2 Sets de datos

Es difícil encontrar sets de datos reales y abiertos que incluyan un gran número de ejemplos o deriva conceptual, por lo tanto, en la literatura, es bastante común encontrarse con la realización de pruebas sobre conjuntos de datos sintéticos. Se pueden encontrar implementaciones libres de los generadores que permiten crear una cantidad potencialmente infinita de ejemplos e incluso introducir deriva conceptual según las especificaciones del usuario, lo que los hace muy útiles en la evaluación de algoritmos de aprendizaje automático en streams.

Para las pruebas que se van a realizar se usarán 4 sets de datos sintéticos: SEA, RandomRBF, LED y Hyperplane. Además, también se incluirán 2 sets de datos reales: Electricity y Shuttle.

Los sets de datos sintéticos se han generado usando la plataforma MOA [10] en su versión 16.04, Electricity se obtuvo de los datasets de la web de MOA[8] y Shuttle del repositorio UCI[15]. El proceso de generación de los datos sintéticos se detalla en el apéndice B.

A continuación se describe brevemente las características de los datos.

SEA. Fue introducido en [44]. Tiene 3 atributos numéricos de los cuales solo los dos primeros son relevantes. Los 3 atributos tienen valores entre 0 y 10. El generador divide los ejemplos en 4 bloques y determina la clase de cada uno mediante una función lineal que separa las 2 clases existentes. Se introduce un 10% de ruido en los ejemplos. Este set de datos incorpora deriva conceptual brusca. Se genera 1000000 ejemplos para las pruebas.

RandomRBF. Genera datos a partir de una función de base radial. Los ejemplos se crean calculando un desplazamiento en una dirección y con magnitud aleatorias respecto de uno de entre varios centroides con una clase asociada. La deriva conceptual se obtiene moviendo los centroides con una velocidad constante. Este set de datos incorpora deriva conceptual gradual que se aplica de forma constante a lo largo de todos los ejemplos. La magnitud de dicha deriva es de 0.00001 para cada ejemplo. Se genera 1000000 ejemplos con 10 atributos numéricos y 5 clases.

LED. Se originó en [12]. Genera un problema basado en predecir el dígito mostrado en un monitor LED de 7 segmentos. Tiene 7 atributos de tipo booleano. Además cada atributo tiene un 10% de posibilidades de resultar invertido. Hay 10 clases posibles y la precisión óptima de Bayes es de un 74%. Se genera 1000000 ejemplos en los que se introduce deriva conceptual.

Hyperplane. Genera ejemplos cuya clase se determina usando un hiperplano como frontera de decisión. La deriva conceptual se introduce cambiando con el tiempo los pesos w_i de la ecuación $\sum_{i=1}^{d} w_i x_i = w_0 = \sum_{i=1}^{d} w_i$ que define al hiperplano de d dimensiones. Este set de datos incorpora deriva conceptual gradual que se aplica de forma constante durante todos los ejemplos. La magnitud de la deriva es de 0.001 para cada ejemplo. Se genera 1000000 ejemplos con 10 atributos numéricos, 2 clases y un 5% de ruido para cada ejemplo.

Electricity. Este set de datos se introduce en [28] y acumula información sobre los precios de la electricidad en New South Wales, Australia. Contiene 45312 instancias con 8 atributos numéricos tomadas a intervalos de 30 minutos. El objetivo es determinar si los precios aumentan o bajan respecto a la media móvil de las últimas 48 horas, por lo tanto hay 2 clases.

Shuttle. Dispone de 58000 ejemplos con 9 atributos numéricos y 7 clases. Aproximadamente un 80% de los ejemplos corresponden a la clase 1. La precisión objetivo es de alrededor de un 99%.

3.3 Configuración experimental

Todas las pruebas realizadas se han llevado a cabo en una máquina equipada con un procesador Intel Core i7-5500U @ 2.4 GHz, 8 GB de memoria RAM y un SO Windows 8.1 de 64 bits.

Los parámetros utilizados para la ELM son: $C = 0.003,\ \varepsilon = 0.0001,\ \tau = 1, c = 5.$ El número de neuronas en cada capa se inicializa a $5 * nº\ de\ atributos$. La función de activación es la sigmoide. El valor de τ elegido se justifica porque es necesario conocer a priori la precisión de la ELM en el set de datos para poder usarse de forma eficaz, de lo contrario se obliga a reentrenar constantemente.

Para el MLP también se usan 2 capas ocultas con un número de neuronas igual a $5 * nº\ de\ atributos$. La función de activación es sigmoide. El optimizador usado es ADAM[31] usando momento de Nesterov, con los parámetros por defecto y la entropía cruzada como métrica.

Tanto ELM como MLP se evalúan siguiendo el procedimiento prequential (descrito en la sección 1.5.2.2) por bloques de datos, incluyendo el uso de un factor de envejecimiento de 0.9. El tamaño de bloque elegido es de 10000 para los datos sintéticos y de 1000 para los reales.

3.4 Resultados

Como se discutido anteriormente, los algoritmos aplicados a streams no solo deben ser precisos sino que además deben cumplir unos requerimientos de memoria y tiempo de ejecución; por lo tanto se discutirán estos 3 aspectos a continuación.

3.4.1 Precisión

En la Tabla 3.1 se muestran las precisiones alcanzadas en los diferentes sets de datos.

Tabla 3.1.- Precisión en los sets de datos

	SEA	RBF	LED	Hyperplane	Electricity	Shuttle
ELM	0.875959263	0.553606475	0.72201487	0.810384401	0.658907005	0.852394052
MLP	0.657126713	0.35589652	0.591646892	0.56775419	0.663855473	0.790048641

Como puede observarse, la ELM alcanza precisiones mayores en los sets de datos sintéticos, mientras que la diferencia se reduce para los datos reales, llegando el MLP a alcanzar una mayor precisión en Electricity. Una posible causa de este fenómeno se puede hallar observando la evolución de la precisión en el tiempo. En el set de datos Hyperplane (Figura 3.1) se tiene que los datos están sometidos a un proceso de deriva conceptual continua, lo que dificulta la formación del modelo. Se puede observar que el MLP tiende a estabilizarse pero alcanzando una precisión menor mientras que la ELM sufre de más oscilaciones pero logra un mejor rendimiento. Esta diferencia puede achacarse, al menos en parte, a que el tamaño de bloque elegido es demasiado grande para poder actualizar con

eficacia el modelo del MLP: El algoritmo de descenso de gradiente utilizado opera mejor cuando se ajusta el modelo con mayor frecuencia.

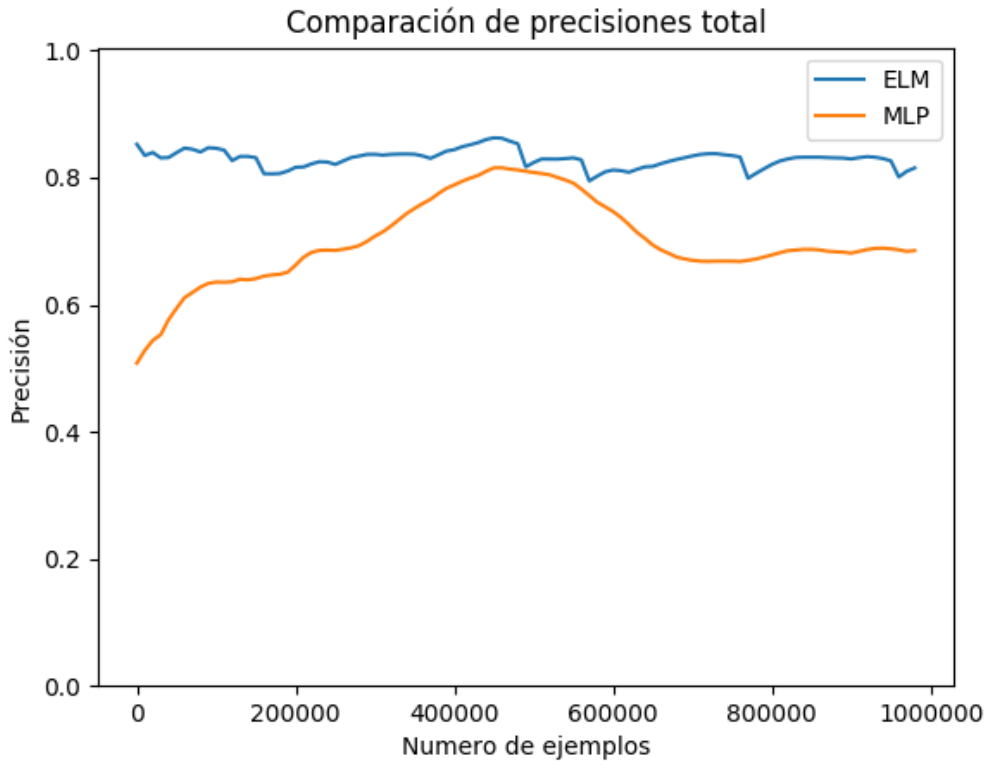

Figura 3.1.- Evolución de la precisión en Hyperplane

La tendencia anterior se observa más claramente en la Figura 3.2, sobre el set de datos RBF. El MLP alcanza un máximo de precisión alrededor de los 200000 ejemplos y se mantiene en ese nivel durante el resto de la evaluación. La ELM va mejorando paulatinamente pero siempre encontrándose con pequeños períodos en los que disminuye su rendimiento debido a la deriva conceptual.

Respecto al set RBF, es interesante como es en el que peores resultados se obtienen a pesar de haber definido una magnitud de deriva bastante pequeña. Estos resultados podrían deberse al sistema de generación de los datos: desviación aleatoria respecto de centroides que determinan las clases, lo que da una distribución complicada de manejar si estos centroides están cercanos en el espacio. En el set usado se definen 50 centroides, todos ellos sometidos a deriva, lo que puede justificar la baja precisión.

En el set Shuttle (en la Figura 3.3) en el que en principio no existe deriva, los resultados no solo son similares sino que también lo es el comportamiento. Las precisiones llegan a solaparse y durante todo el test mantienen perfiles similares. Sin embargo la precisión global es baja respecto al 99% que se espera.

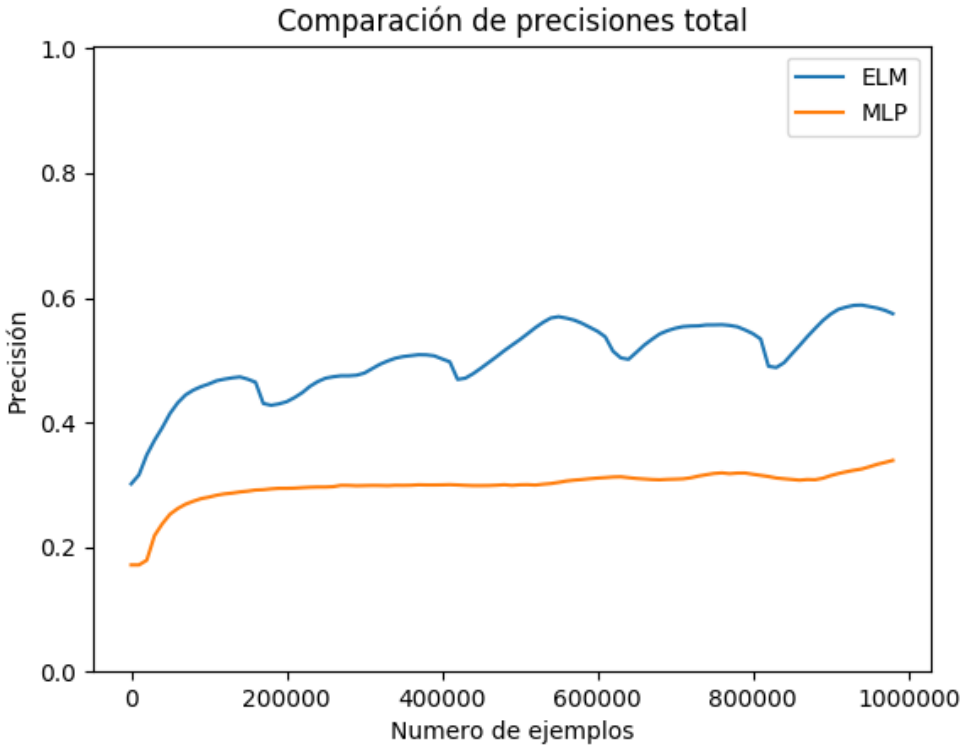

Figura 3.2.- Evolución de la precisión en RBF

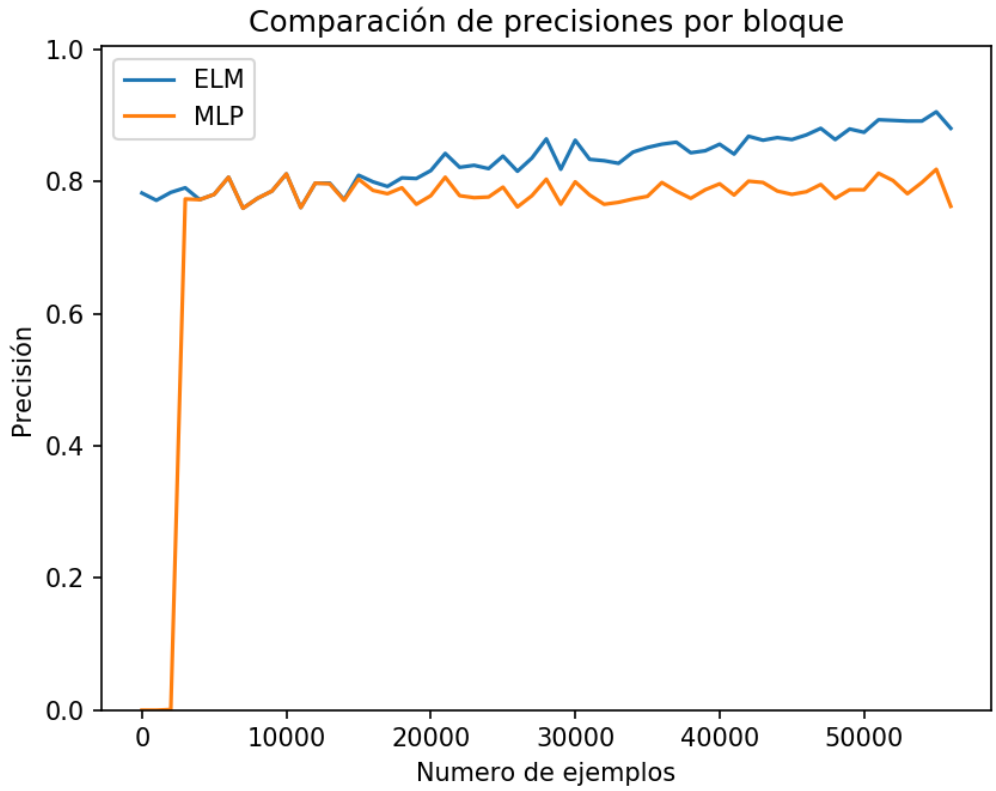

Figura 3.3.- Precisión por bloque de datos en Shuttle

El set de datos Electricity (ilustrado en la Figura 3.4), sin embargo, sí que contiene deriva conceptual, tal y como se apunta en [48] y además existe una dependencia temporal en los datos. Con el tamaño de bloque elegido de 1000, no es posible captar la deriva que se produce, ya que en el set de datos un día tiene 48 observaciones. La situación encontrada, por tanto, es el problema asociado al uso de ventanas de ejemplos (discutido en el epígrafe 1.3.1): ajustar el tamaño de forma correcta. El resultado se ve en la: la evolución de la precisión es irregular y es mucho más baja que la señalada en la referencia. Al seleccionar un tamaño de ventana más pequeño, de 100, (Figura 3.5) se observa una mejora notable en la precisión.

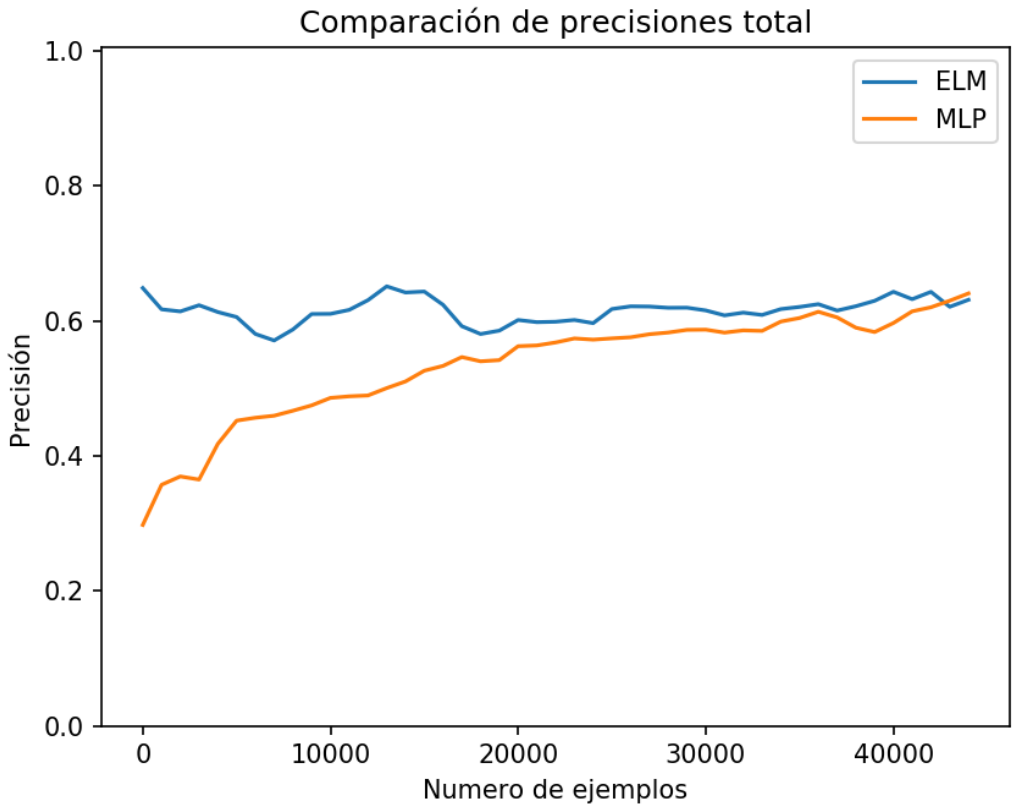

Figura 3.4.- Evolución de la precisión en Electricity

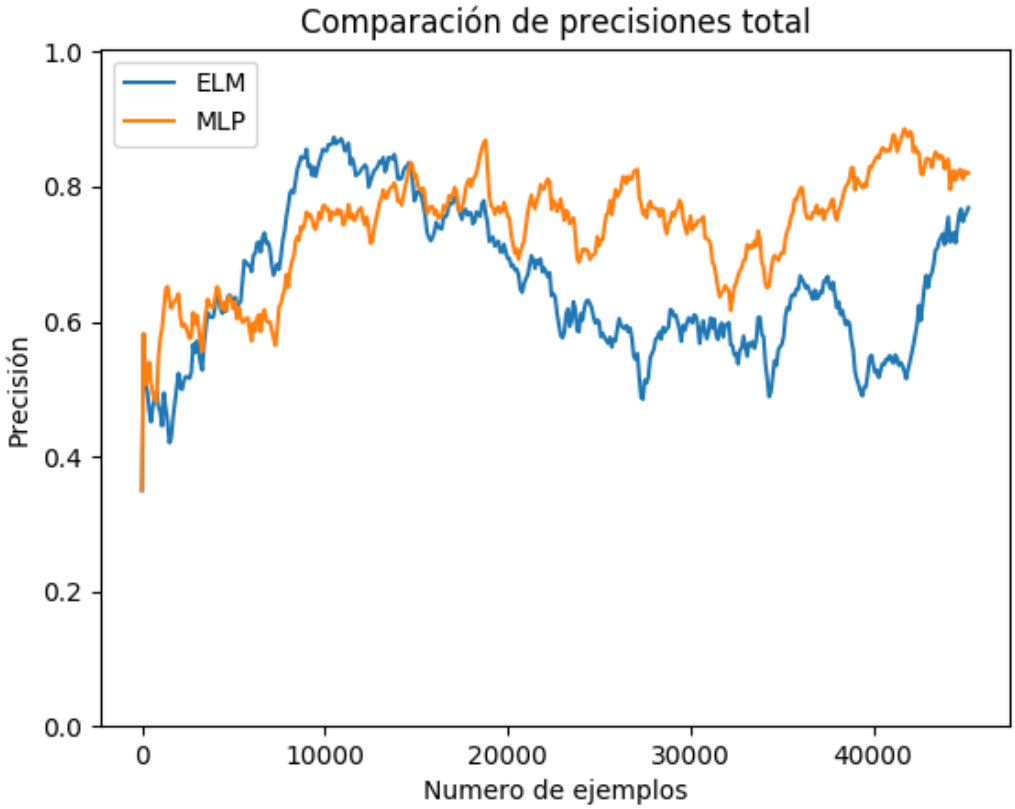

Figura 3.5.- Precisión en Electricity con tamaño de bloque 100

Finalmente, los sets SEA y LED, mostrados en la Figura 3.7 y Figura 3.6 respectivamente, tienen un comportamiento un tanto extraño. En ambos se introduce deriva conceptual, sin embargo la evolución de la precisión tanto para la ELM como para el MLP llega a ser prácticamente constante, por lo que la deriva realmente no tiene impacto visible sobre el rendimiento.

En la documentación de MOA se describe la generación de SEA a partir de 4 fuentes con diferentes umbrales, así que es posible que el mecanismo usado no haya podido introducir cambios relevantes o que los datos sean lo suficientemente fáciles de modelar (la clase de los datos se determina mediante una desigualdad simple tal y como se menciona en la sección 3.2) como para que el efecto no se note. Por otra parte, los autores del set LED afirman que el problema de clasificación sería fácil de no ser por la presencia de ruido así que esa podría ser la causa del comportamiento observado.

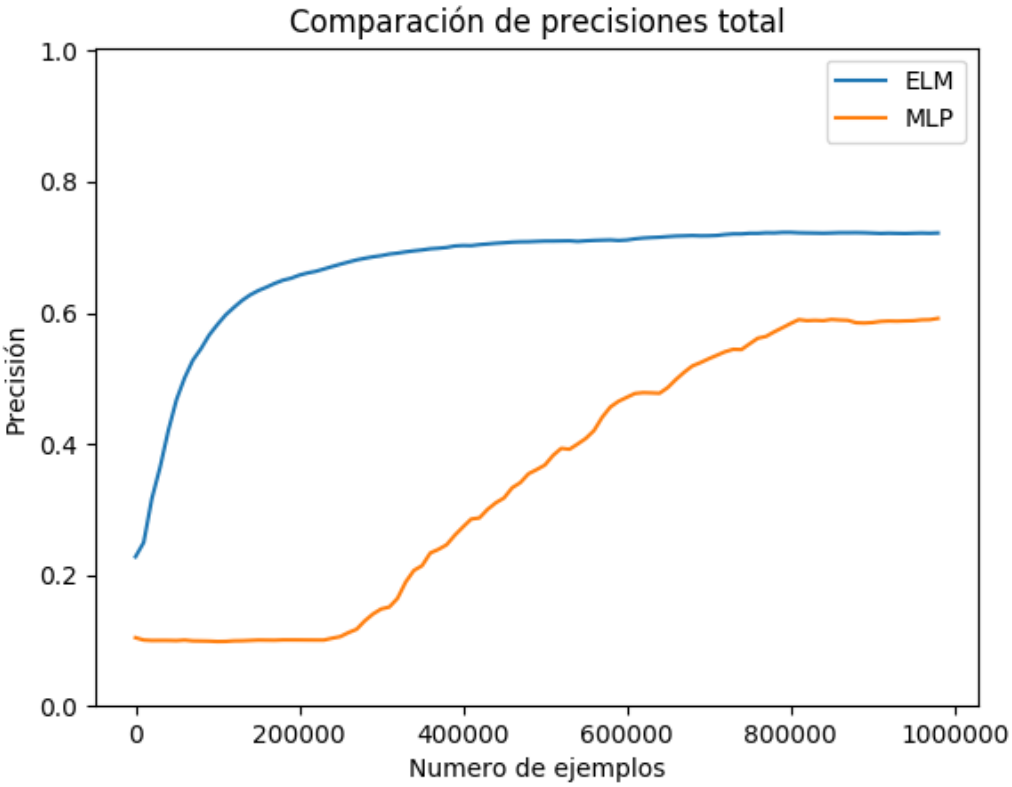

Figura 3.6.- Evolución de la precisión en LED

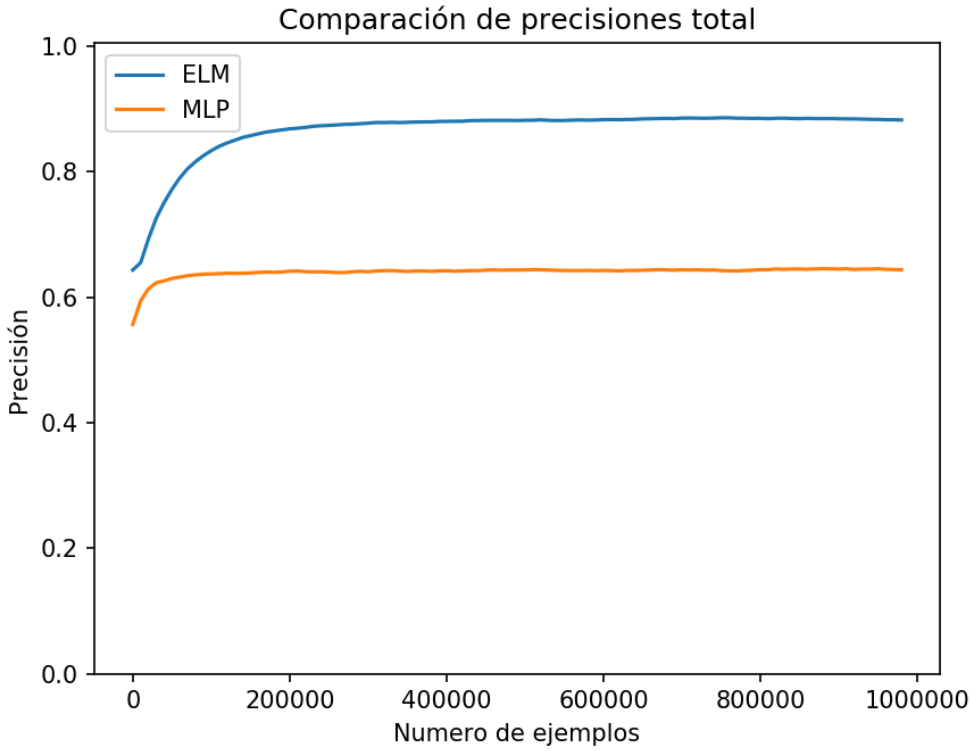

Figura 3.7.- Evolución de la precisión en SEA

3.4.2 Memoria

Los resultados mostrados en la Tabla 3.2 indican que la cantidad de memoria consumida es similar en todos los sets de datos. Mientras que el MLP tiene un consumo casi constante, existe más variabilidad en la ELM. Para ambos el consumo es razonable, comparable al de un editor de texto como Word, así que no debería suponer un problema para un equipo moderno.

Tabla 3.2.- Consumo de memoria (en MB) en los sets de datos

	SEA	RBF	LED	Hyperplane	Electricity	Shuttle
ELM	102.340041	120.2929688	111.6774779	119.7743845	104.9322049	102.0024671
MLP	122.0968339	126.9757734	125.4334359	125.7304293	126.3243924	122.0968339

La ELM consume más memoria en los sets de datos en los que detecta deriva conceptual. Por ejemplo, en la Figura 3.8 se muestra el consumo de memoria en RBF.

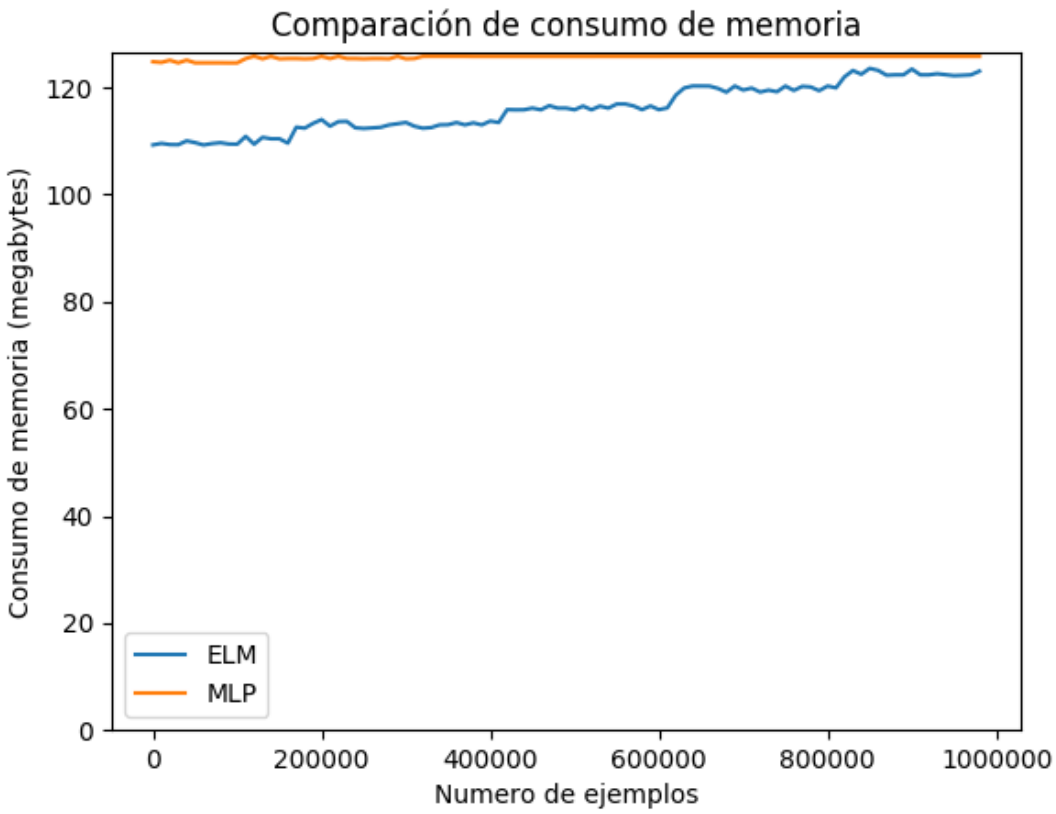

Figura 3.8.- Evolución del consumo de memoria en RBF

Se observa que el incremento de memoria coincide con los puntos en los que se reduce la precisión en la Figura 3.2. Esto se debe a que la ELM incrementa el número de neuronas que posee al detectar deriva conceptual, lo que incrementa el espacio de almacenamiento requerido. No obstante, el incremento del número de neuronas está limitado, como ya se explicó en la descripción del

algoritmo en la sección 2.5.1, por lo que el consumo de memoria no puede incrementarse sin control.

Para datos en los no se detecta deriva como Shuttle, el consumo permanece constante. La Figura 3.9 ilustra dicho comportamiento.

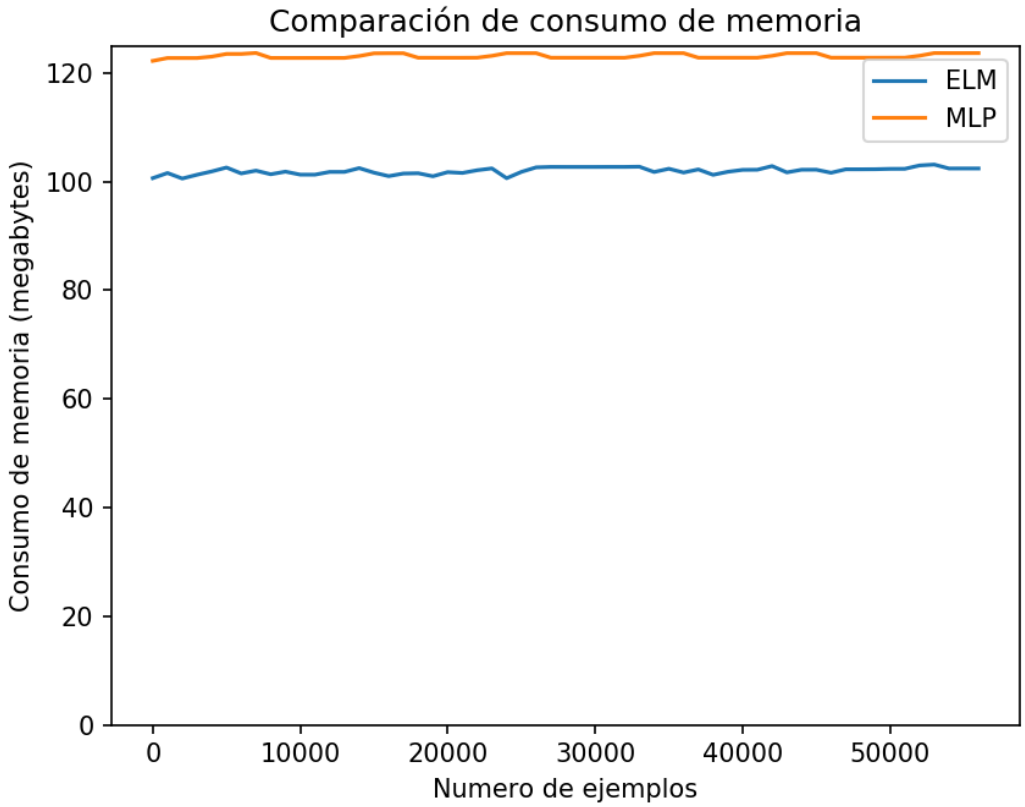

Figura 3.9.- Evolución del consumo de memoria en Shuttle

Se puede ver que ambos sistemas tienen un consumo de memoria constante, por lo que cumplen con uno de los requerimientos de los algoritmos aplicados a streams. Esto es interesante en el caso del MLP, ya no está planteado específicamente para cumplir con las especificaciones dadas, lo que muestra lo bien que se prestan las redes neuronales a este tipo de computación, tal y como se comentó en el epígrafe 2.5.

3.4.3 Tiempo de ejecución

Tal y como se observa en la Tabla 3.3, la ELM es más rápida que el MLP en todos los sets de datos seleccionados. Las diferencias en tiempos de ejecución entre los datos sintéticos pueden achacarse a las diferencias en dimensionalidad; SEA solo tiene 3 atributos númericos y LED 7 booleanos mientras que RBF e Hyperplane tienen 10 cada uno. Sin embargo, este no es el único factor, como se verá a continuación.

Tabla 3.3.- Tiempo de ejecución (en segundos) en los sets de datos

	SEA	RBF	LED	Hyperplane	Electricity	Shuttle
ELM	20.92124028	74.31981149	26.12328318	75.23870395	2.828800773	1.675437062
MLP	33.95028835	104.4042859	55.24048	92.40098018	4.978643717	4.236073561

Al igual que ocurría con el consumo de memoria, el tiempo de ejecución de la ELM se ve impactado por la deriva conceptual. Al detectar cambio en los datos no solo se incrementa el número de neuronas sino que además se reentrena, lo que tiene un impacto negativo en el rendimiento. Esto se puede apreciar fácilmente comparando la evolución del tiempo de ejecución para los sets de datos SEA y RBF, ilustrados en la Figura 3.10 y Figura 3.11 respectivamente.

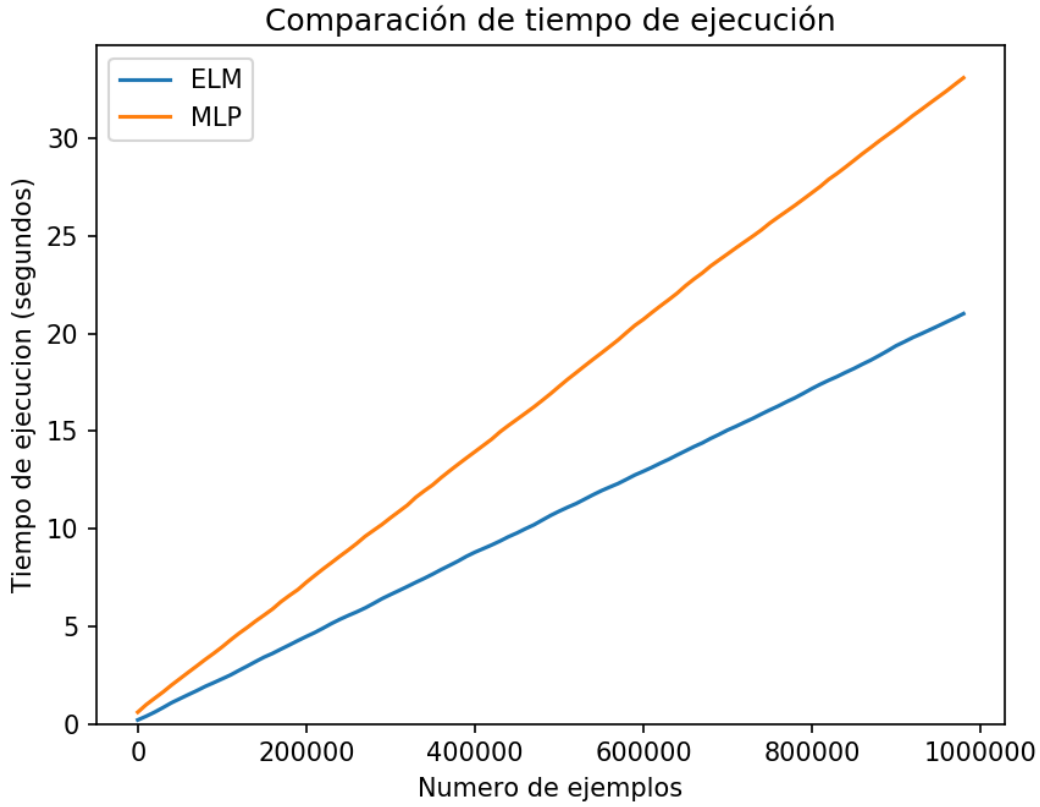

Figura 3.10.- Tiempo de ejecución en SEA

En SEA no se detectaba deriva conceptual por lo que el tiempo de ejecución es aproximadamente lineal para la ELM. La linealidad también se aplica para el MLP.

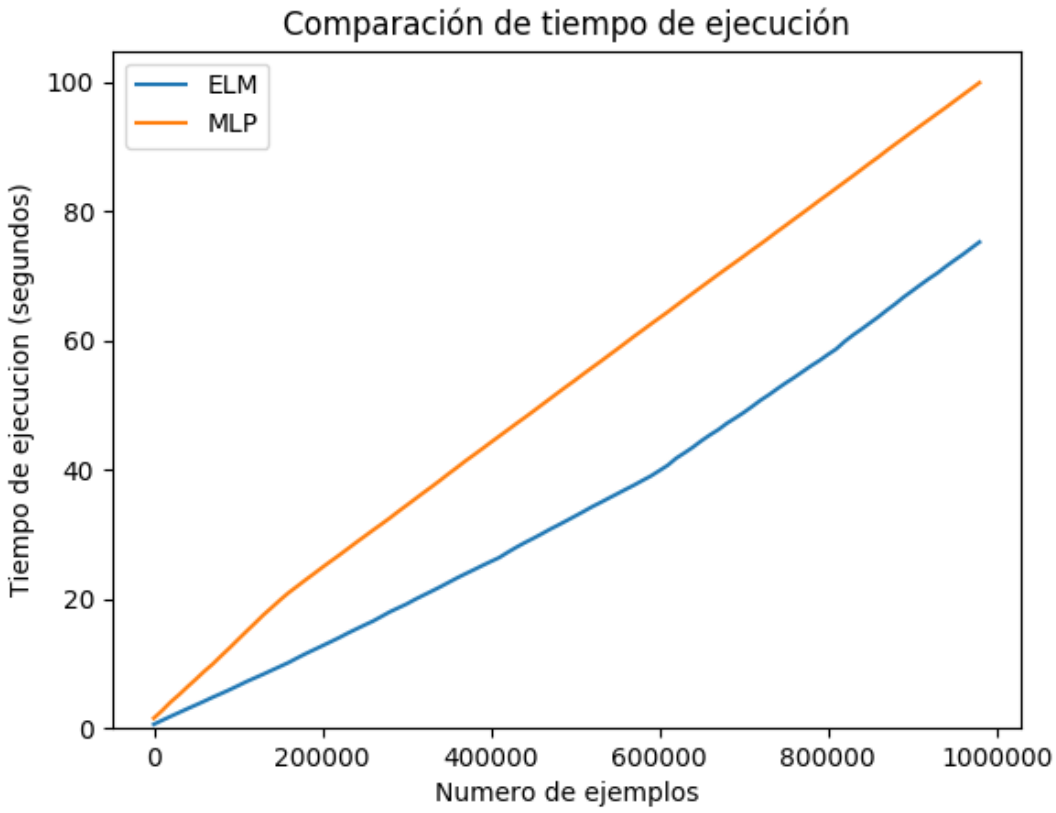

Figura 3.11.- Evolución del tiempo de ejecución en RBF

En el caso de RBF, sin embargo, se puede apreciar cómo cambia la pendiente del tiempo de ejecución de la ELM en torno a los 600000 ejemplos procesados. El resultado sigue siendo lineal, eso sí, pero el incremento es notable. Si el set de datos usado fuese más grande o incorporase un grado mayor de cambio no se podría descartar que la constante detección de deriva acabase por incrementar la pendiente hasta superar al MLP.

Las tendencias observadas se mantienen para el resto de datos: MLP con un tiempo lineal, al igual que ELM, pero esta última ve incrementado el tiempo de ejecución al ocurrir deriva conceptual.

Por último, es necesario resaltar que la complejidad temporal lineal es otro requisito de la clasificación en streams que satisfacen ambos sistemas.

3.4.4 Impacto del parámetro C

En los cálculos expuestos en la descripción del algoritmo de la DELM en la sección se añadían valores positivos a la diagonal de la matriz $H^T H$ mediante una matriz $\frac{I}{C}$. Se supone que está adición incrementa la estabilidad de la solución y proporciona una mayor capacidad de generalización al ajustar mejor el modelo a los datos de ejemplo. Para comprobar el impacto práctico de este parámetro se testea el rendimiento de la ELM sobre los sets de datos usados para diferentes

valores de C. El resto de parámetros son los mismos que en las anteriores pruebas.

En la Tabla 3.4 se muestran las diferencias en precisión para los diferentes valores de C. Se omiten el consumo de memoria y el tiempo de ejecución por ser extremadamente similares a los valores obtenidos anteriormente.

Tabla 3.4.- Precisión de la ELM para distintos valores de C

	SEA	RBF	LED	Hyperplane	Electricity	Shuttle
C=0.003	0.882653827	0.545102866	0.725841674	0.800229324	0.669988139	0.829188542
C=0.05	0.889193454	0.711616266	0.735349899	0.83840682	0.761723016	0.966670433
C=1	0.886633448	0.744354142	0.739775045	0.809024652	0.755904351	0.986616014
C=5	0.881441942	0.75023827	0.740274651	0.819221511	0.748920691	0.990263813

Los resultados son variados. En primer lugar hay que destacar que en los casos en los que se produce una mejora respecto al valor por defecto de C (0.003), las diferencias entre los demás son mucho menos significativas. En los sets de datos en los que no se incrementa la precisión con distintos valores de C no se produce ninguna disminución del rendimiento, pero sí que se obtiene la prometida mejora en la estabilidad del modelo como se verá más adelante. La conclusión que se obtiene es que C=0.003 es un valor excesivamente pequeño para ser usado por defecto con los datos examinados.

La precisión en SEA, ilustrada en la Figura 3.12, no mejora significativamente y en LED el incremento es muy pequeño. Esto puede aducirse a que ya con C=0.003 la precisión para ambos datasets llegaba a un punto en el que permanecía constante, lo que quiere decir que se ha llegado al límite de la capacidad de la ELM y por tanto un mayor grado de ajuste mediante el incremento de C no resulta relevante.

A pesar de que la precisión global no se incrementa, se puede observar en la figura que valores mayores de C cumplen con la promesa de una mayor estabilidad, ya que la precisión es constante durante casi todo el test. Esta estabilidad también se aplica al set LED. Además, cabe destacar que se detecta deriva conceptual para los valores mayores de 0.003 (el por defecto) mientras que originalmente no se apreciaba, ya que un valor de C bajo aumenta la tolerancia al error. Esto apoya la hipótesis de que SEA es un problema fácil para la ELM, puesto que con un ajuste relativamente débil se obtiene la máxima precisión posible.

El caso de Hyperplane, mostrado en la Figura 3.13, es bastante particular puesto que no hay diferencias significativas ni en la precisión ni en la evolución de la precisión en el tiempo. Esto puede producirse porque en este set solo existen 2 clases y la pertenencia de un ejemplo a una u otra se define a partir del

hiperplano generado, así que al estar bien definida la separación entre clases el impacto de C en el rendimiento es menor.

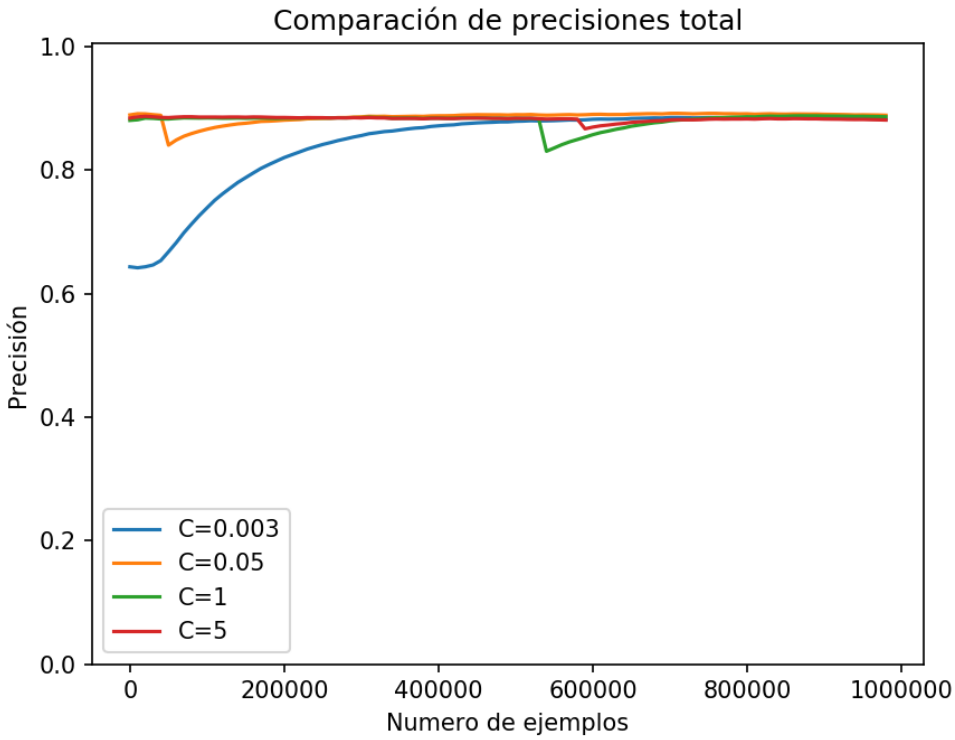

Figura 3.12.- Precisión en SEA para distintos valores de C

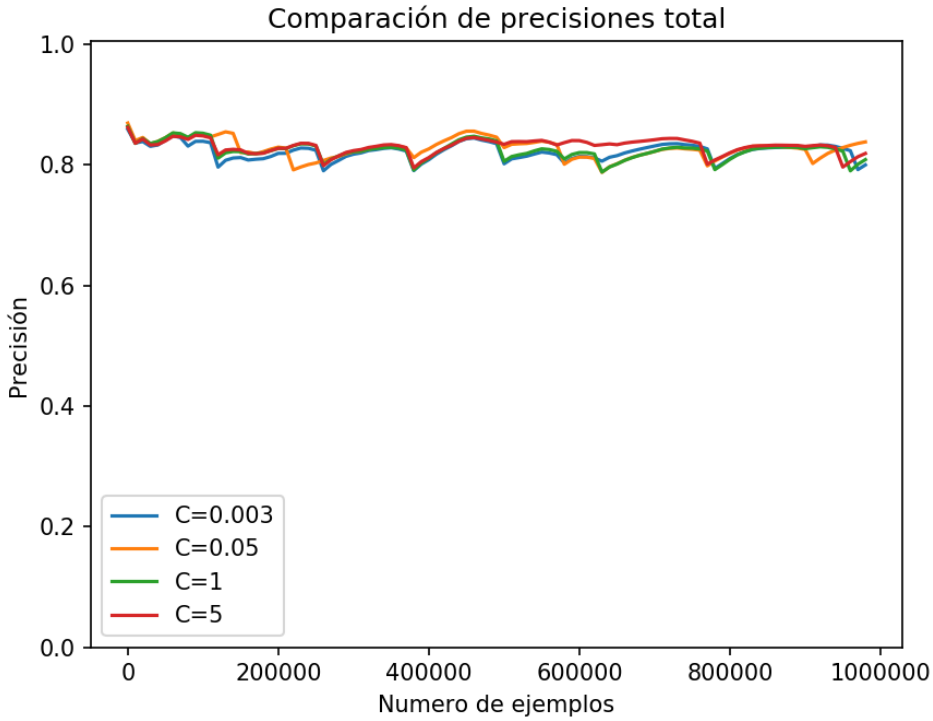

Figura 3.13.- Precisión en Hyperplane para diferentes valores de C

En RBF, Electricity y Shuttle se produce un incremento sustancial de la precisión.

Para Electricity (Figura 3.14) la precisión mejora pero en menor medida, aunque el perfil de la precisión se suaviza. En el caso de Shuttle, la precisión toma una forma similar a la de LED o SEA: una constante. Para RBF (Figura 3.15) este incremento en la precisión también lleva asociado una mejora en la detección de deriva conceptual; el ajuste más preciso del modelo a los datos incrementa la sensibilidad de la ELM a los cambios.

Finalmente, hay resaltar que la mejora de la precisión al aumentar C tiene un carácter asintótico. Como se puede observar en la Figura 3.15, las diferencias entre los valores de 0.005, 1 y 5 son bastante pequeñas, por lo que cabe esperar que incrementar aún más el parámetro no produzca resultados significativos.

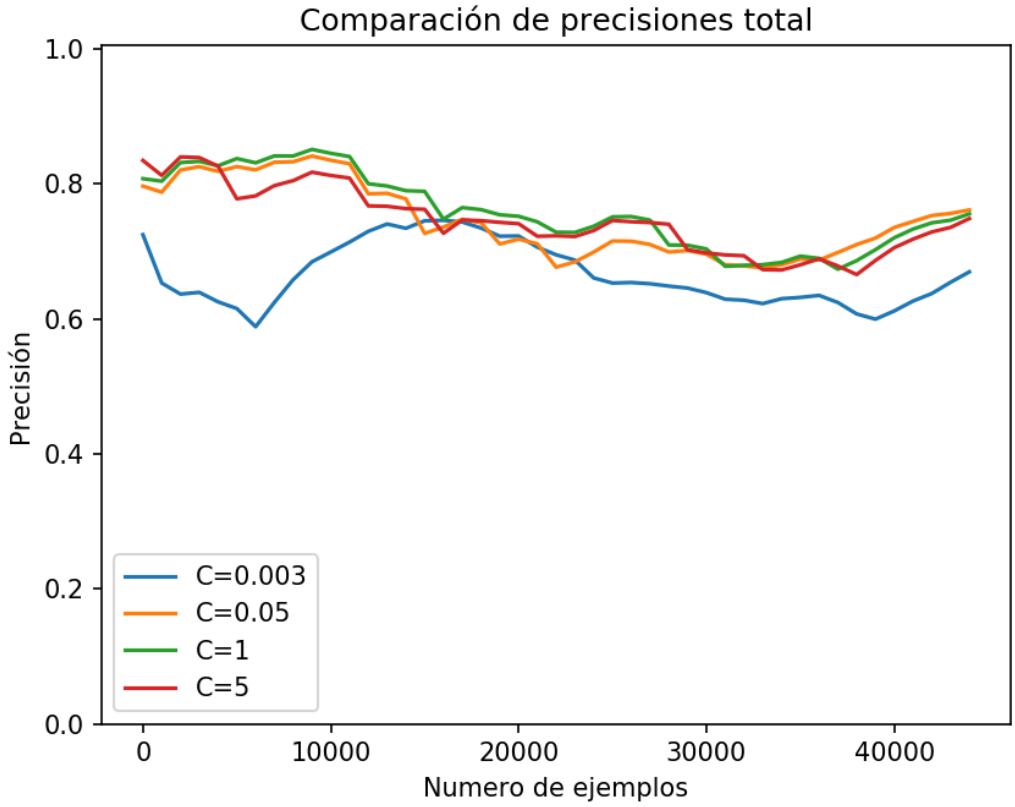

Figura 3.14.- Precisión en Electricity para distintos valores de C

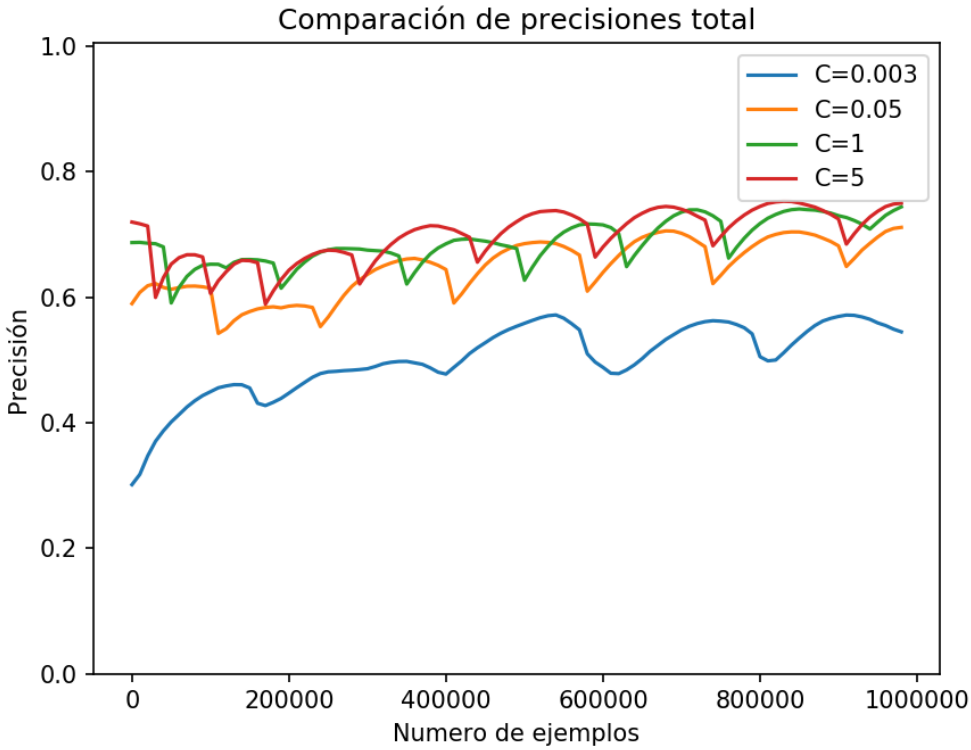

Figura 3.15.- Precisión en RBF para distintos valores de C

3.4.5 Impacto de la deriva conceptual

La adaptación al cambio es uno de los mayores retos a superar para los algoritmos aplicados a streams. Para comprobar como varía la precisión de la ELM en función de la deriva conceptual presente en los datos, se lleva a cabo una serie de pruebas con deriva de diferente magnitud.

Para las pruebas se seleccionan los datasets Hyperplane y RBF.

En base a los resultados del epígrafe anterior se escoge un valor de $C = 1$. Todos los sets de datos usados son de 1000000 ejemplos. El resto de parámetros son iguales a los de pruebas anteriores. Los resultados se muestran en la Tabla 3.5.

Tabla 3.5.- Precisión de la ELM para distintas magnitudes de deriva

	Hyperplane	RBF
magnitud= 0	0.919532379	0.756620339
magnitud= 0.00001	0.895491568	0.749516886
magnitud= 0.001	0.814362383	0.30118478
magnitud= 0.1	0.827525325	0.304456281

Una de las primeras observaciones que pueden realizarse es que el impacto de la deriva en los dos sets de datos es bastante diferente. La reducción de la precisión en RBF para mayores magnitudes de deriva es mucho mayor que para Hyperplane (reducción de 0.45 y 0.1 respectivamente). Estos resultados están en

consonancia con lo comentado anteriormente respecto a la complejidad de RBF y Hyperplane.

También se pueden encontrar ocurrencias interesantes al analizar los perfiles de la precisión. En Hyperplane (Figura 3.16) las precisiones siguen una evolución más o menos normal, con la excepción de que la precisión total es mayor para el valor de magnitud 0.1 que para 0.001. Otra anormalidad es la detección de deriva conceptual alrededor de los 300000 ejemplos en el caso en el que la

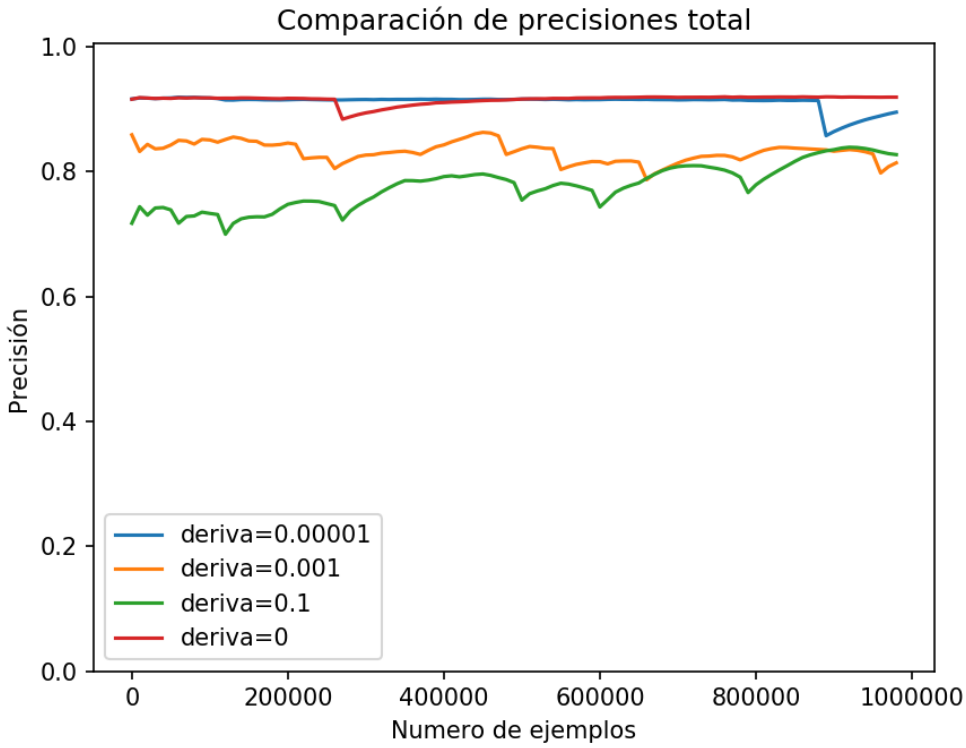

Figura 3.16.- Precisión en Hyperplane para diferentes magnitudes de deriva

magnitud de esta es 0. Ambas situaciones pueden deberse a que la ELM sacrifica estabilidad para lograr una mejor adaptación a los cambios, por lo que pueden ocurrir irregularidades como las descritas.

Para RBF, ilustrado en la Figura 3.17, la precisión global acaba igualada para el caso sin deriva y el de magnitud 0.00001. Este último es un valor muy pequeño, pero puede observarse que tiene un impacto no despreciable en la precisión, aunque acabe adaptándose. Lo más interesante es que para los valores de 0.001 y 0.1 no se detecta deriva en absoluto a pesar de ser de una magnitud considerable; la causa de esto puede ser el tamaño de bloque elegido. Como se ha discutido, un mayor tamaño de bloque disminuye las capacidades de detección del cambio, y al ser en este caso tan rápido, la ELM es incapaz de descubrirlo por lo que la precisión es más o menos constante y muy baja.

El consumo de memoria y el tiempo de ejecución mantienen las tendencias descritas en apartados 3.4.2 y 3.4.3: los mecanismos de adaptación a la deriva provocan un incremento en ambos parámetros. La Figura 3.18 muestra esta ocurrencia para el tiempo de ejecución y la Figura 3.19 para el consumo de memoria, ambas sobre Hyperplane. Es notable como la proporción del incremento de memoria es menor que la del tiempo.

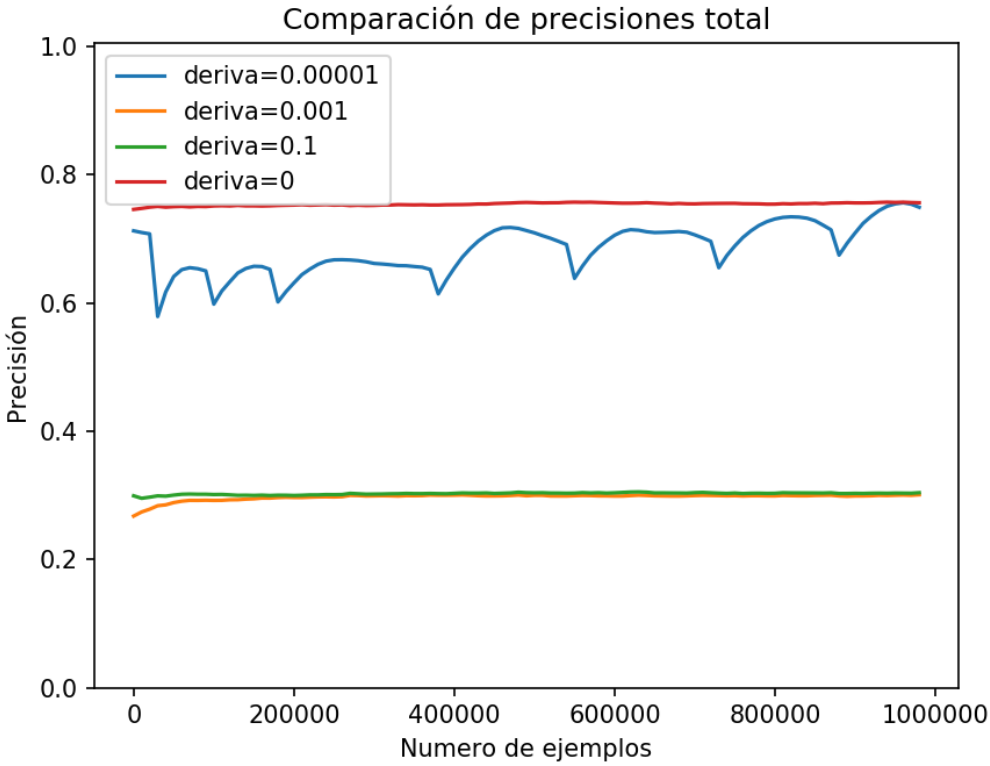

Figura 3.17.- Precisión en RBF para distintas magnitudes de deriva

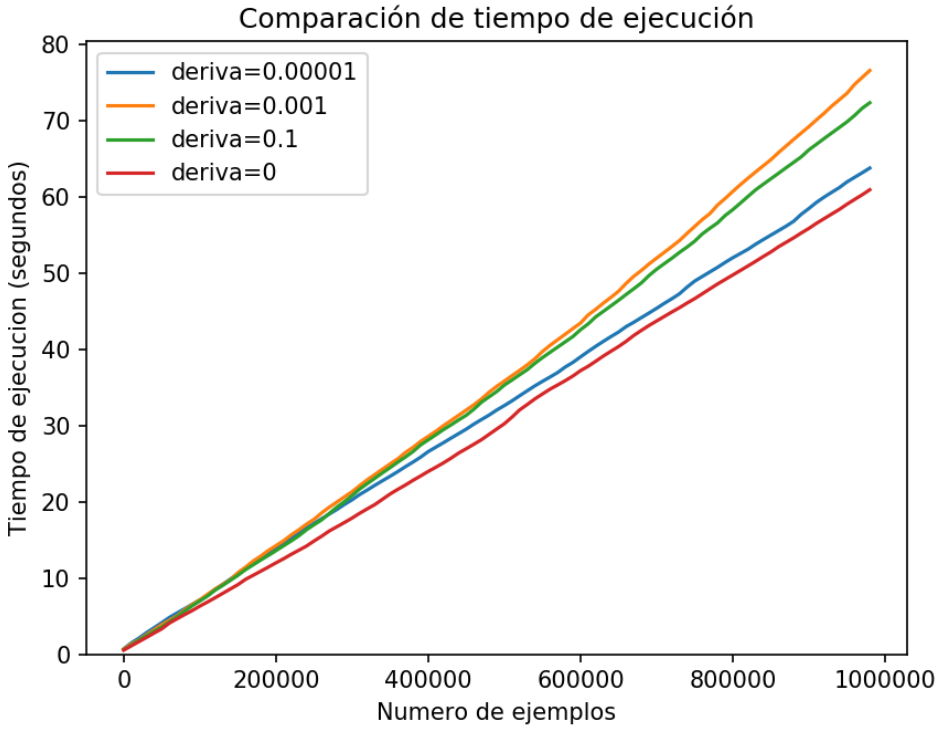

Figura 3.18.- Tiempo de ejecución en Hyperplane para distintas magnitudes de deriva

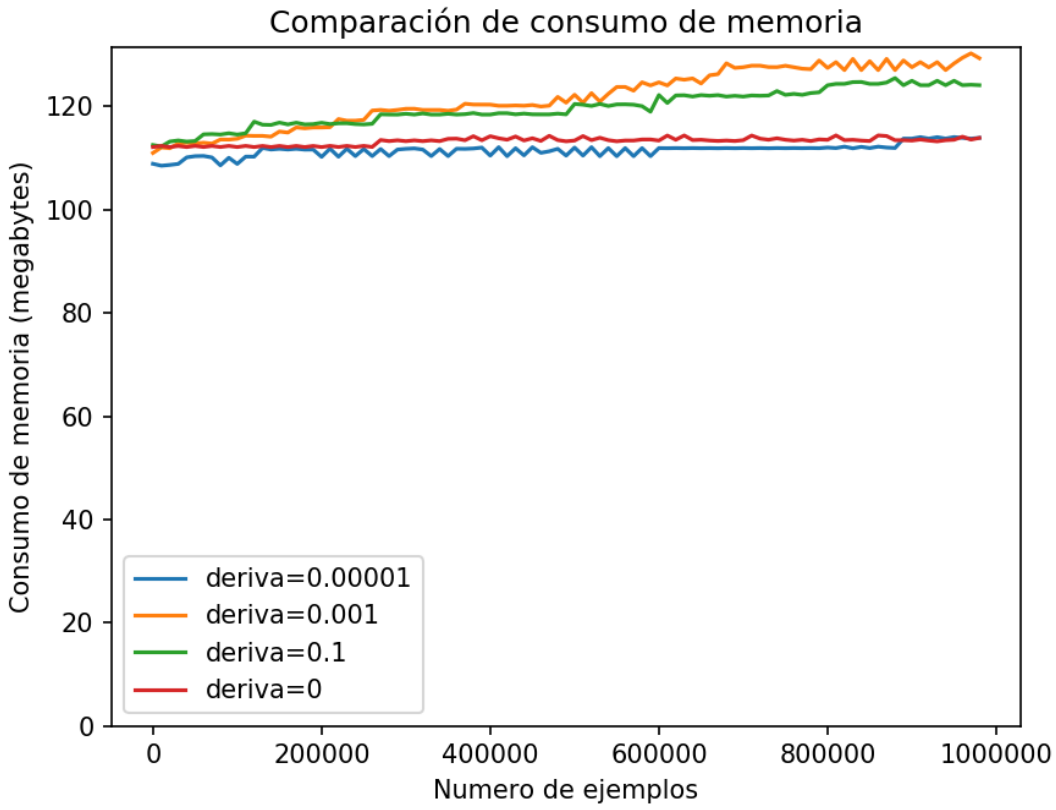

Figura 3.19.- Consumo de memoria en Hyperplane para distintas magnitudes de deriva

3.4.6 Impacto del tamaño de ventana con deriva conceptual

En vista de los resultados obtenidos en el epígrafe 3.4.5 en la precisión en el set RBF se realizan la mismas pruebas pero cambiando esta vez el tamaño de bloque seleccionado. Se usarán tamaños de 1000 y 100 para determinar su relevancia en el rendimiento de la ELM cuando existe deriva conceptual.

Tabla 3.6.- Precisión de la ELM con tamaño de bloque 1000

	Hyperplane	RBF
magnitud= 0	0.918933987	0.750589336
magnitud= 0.00001	0.914895232	0.837255698
magnitud= 0.001	0.881032952	0.33239645
magnitud= 0.1	0.890832134	0.31386136

Los resultados para el tamaño de bloque 1000 se reflejan en la Tabla 3.6. Se observa cómo se produce una mejora para Hyperplane que acerca los resultados a aquellos para los que no existe deriva en absoluto, confirmando que un bloque más pequeño mejora la adaptación al cambio. Esto se ilustra en la Figura 3.20.

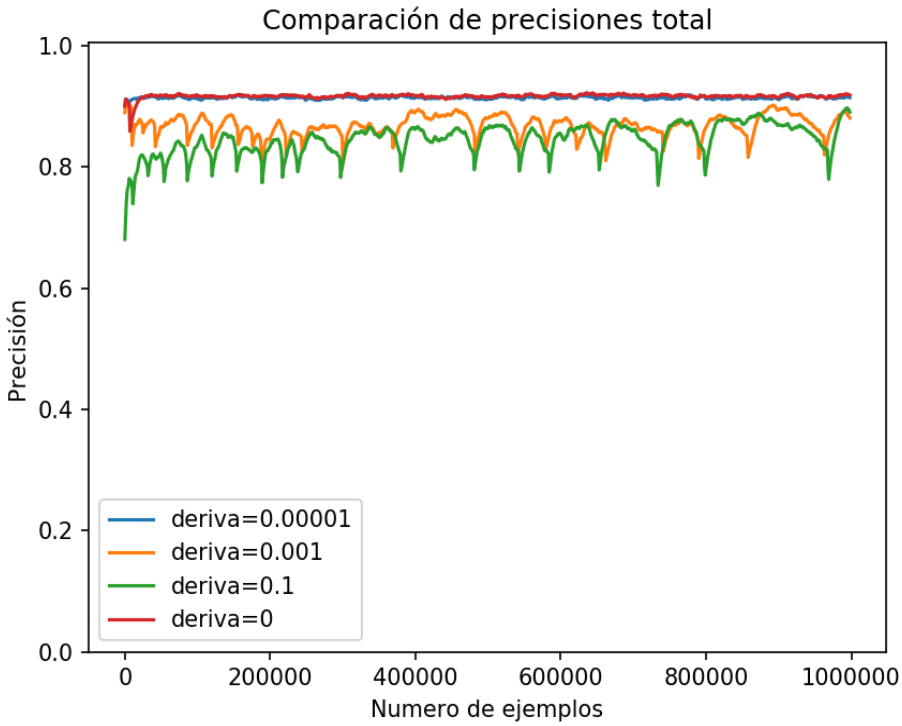

Figura 3.20.- Precisión en Hyperplane con tamaño de ventana 1000

Para RBF también se observa una mejora en el caso de magnitud 0.00001, pero en los otros 2 no hay mejora alguna. Este se debe a que, como ya se ha discutido, la generación de datos en RBF magnifica el impacto de la deriva que se introduce. Para 0.1 y 0.001 el volumen del cambio es suficiente como para que la reducción del bloque no ofrezca mejora alguna.

Si se analiza el tiempo de ejecución, mostrado en la Figura 3.21, se ve como para una magnitud de cambio de 0.001 en RBF el tiempo de ejecución se dispara. La causa de esto es que el tamaño de bloque se ha reducido suficiente como para detectar la ocurrencia de deriva pero sigue siendo demasiado grande como para poder adaptarse con eficacia, como se puede ver con las precisiones obtenidas.

Esta ocurrencia se puede ver mejor observando las precisiones obtenidas para cada bloque de datos procesado. La precisión obtenida oscila constantemente pero no va mejorando como ocurre con el valor de magnitud 0.00001. La Figura 3.22 muestra la ocurrencia anterior.

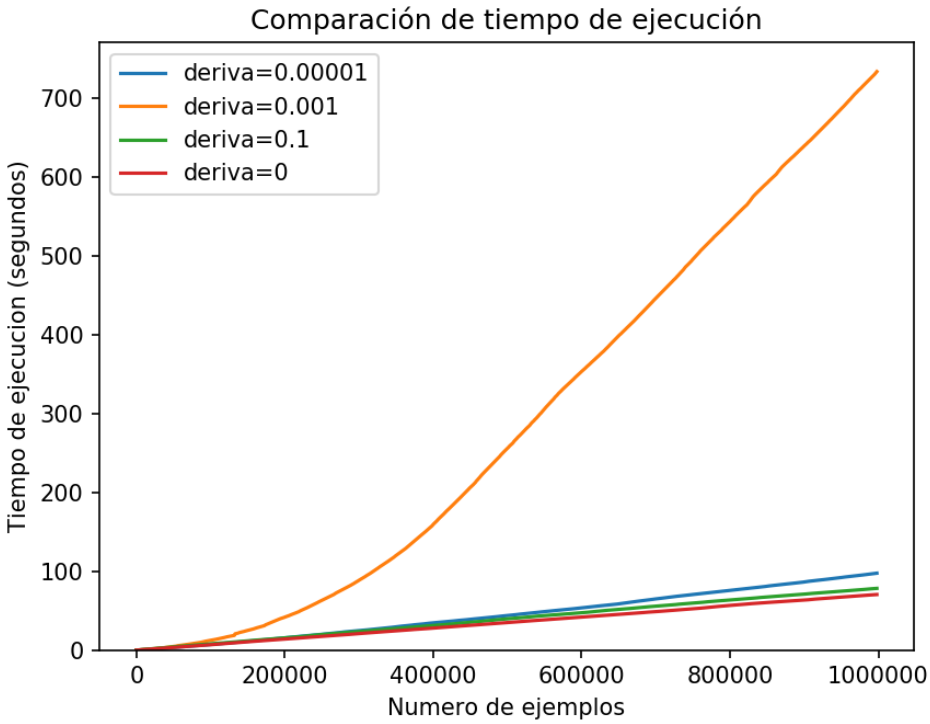

Figura 3.21.- Tiempo de ejecución en RBF con tamaño de bloque 1000

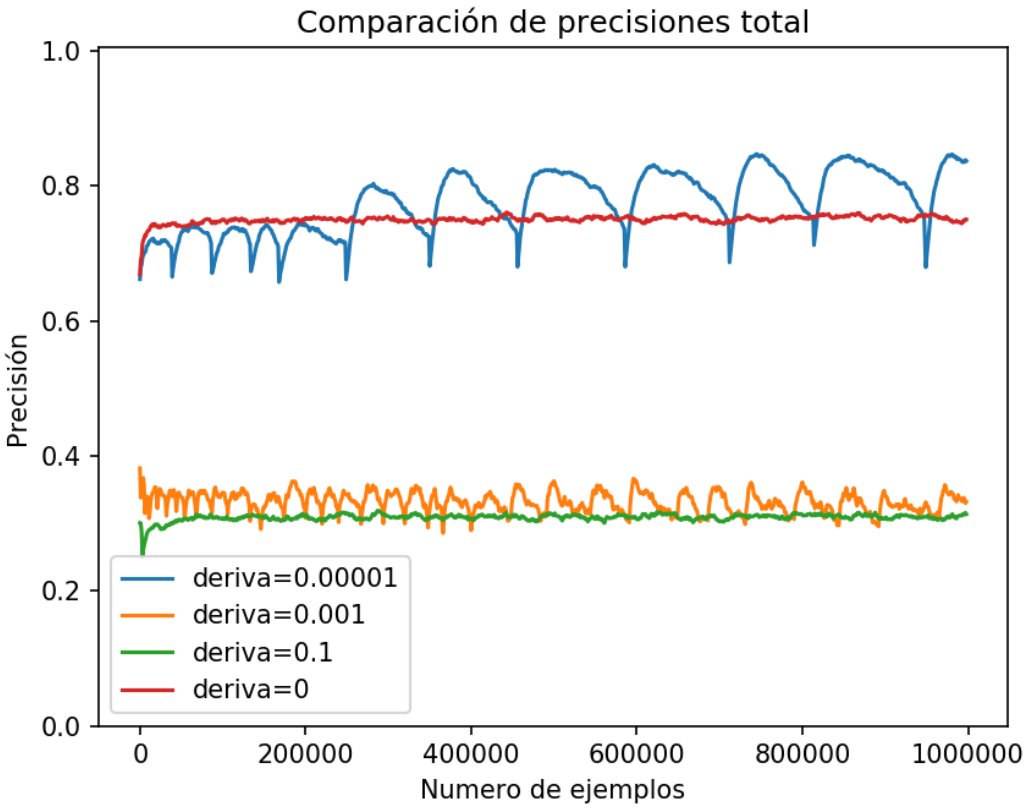

Figura 3.22.- Precisión por bloque en RBF con tamaño de bloque 1000

Para el tamaño de bloque 100 los resultados se presentan en la Tabla 3.7.

Tabla 3.7.- Precisión de la ELM con tamaño de bloque 100

	Hyperplane	RBF
magnitud= 0	0.91771328	0.7459384
magnitud= 0.00001	0.91752293	0.79325115
magnitud= 0.001	0.86365583	0.49554935
magnitud= 0.1	0.8671718	0.31368788

No hay demasiada variación realmente. En Hyperplane las cifras son similares, con una precisión algo menor para las magnitudes 0.001 y 0.1.

La misma similitud se halla en RBF, con la diferencia de que en este caso se observa una mejora para la magnitud de 0.001, que además se ha ejecutado en un tiempo similar que el de los demás valores de deriva. La magnitud de 0.1 vuelve a obtener una precisión igual, por lo que es probable que simplemente sea demasiado elevada como para poder ser procesada de forma adecuada. Esto se muestra en la Figura 3.23.

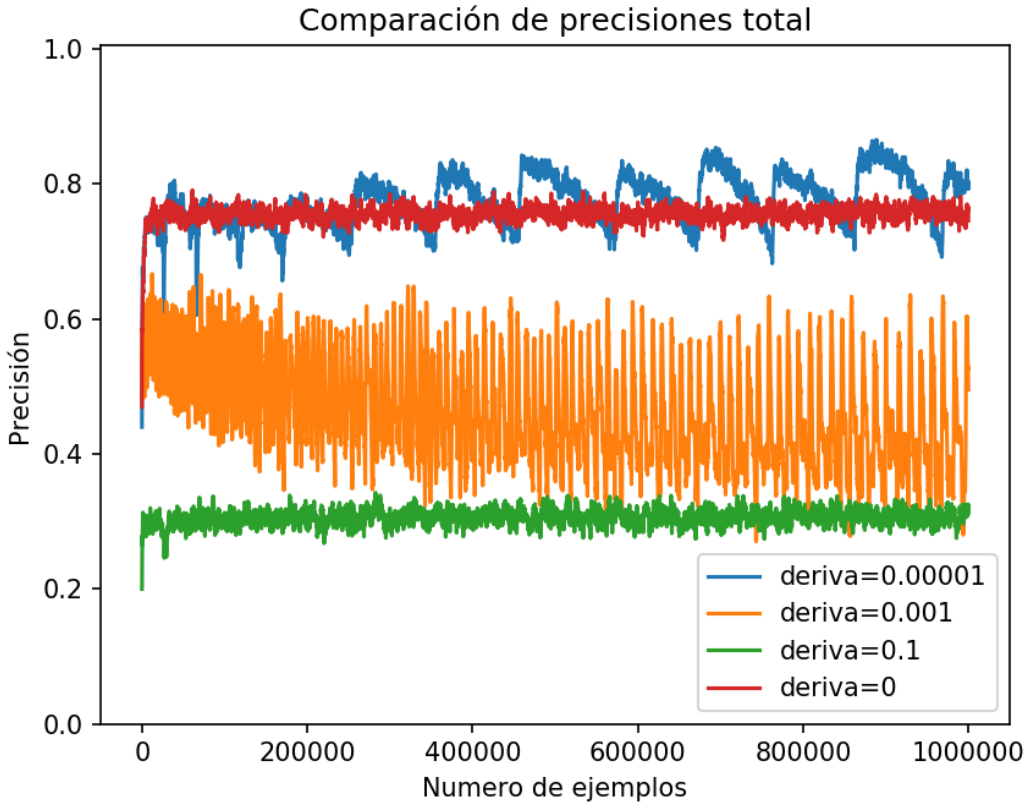

Figura 3.23.- Precisión en RBF con tamaño de bloque 100

Se mencionó en el epígrafe 3.4.5 que la ELM sacrifica estabilidad para poder adaptarse a los cambios. La Figura 3.23 proporciona un buen indicador de este comportamiento. Las oscilaciones en precisión son de mayor magnitud que las que se producían en la Figura 3.17, por lo que se puede afirmar que un tamaño de ventana más pequeño incrementa la sensibilidad a los cambios en los datos, ya que las actualizaciones al modelo se realizan con mayor frecuencia.

Como nota final hay que destacar que al elegir tamaños de bloque menores se incrementa el tiempo de ejecución para todos los valores de deriva. Este incremento se puede observar en la Tabla 3.8 y Tabla 3.9. Este aumento procede simplemente de se realiza un mayor número total de operaciones para procesar todos los datos.

Tabla 3.8.- Tiempo de ejecución (en segundos) de la ELM con tamaño de bloque 1000

	Hyperplane	RBF
magnitud= 0	72.020663	70.6300323
magnitud= 0.00001	69.522476	97.7049453
magnitud= 0.001	120.277037	733.961792
magnitud= 0.1	118.160497	78.448089

Tabla 3.9.- Tiempo de ejecución (en segundos) de la ELM con tamaño de bloque 100

	Hyperplane	RBF
magnitud= 0	120.703706	121.65492
magnitud= 0.00001	126.165398	138.186264
magnitud= 0.001	142.996089	143.939431
magnitud= 0.1	142.043347	138.92167

Conclusiones

Los algoritmos de aprendizaje aplicados a streams resultan bastante ventajosos. El sistema evaluado, la DELM, alcanza una precisión comparable, superior de hecho con los parámetros establecidos, a la de un sistema tradicional como es un perceptrón multicapa. Además ofrece unos tiempos de ejecución y un consumo de memoria menores. Las diferencias, aunque no extremadamente grandes, son lo suficientemente perceptibles como para poder justificar el uso de la DELM incluso para labores de clasificación en un entorno tradicional por lotes. Basándose en los resultados presentados en las referencias, esta misma conclusión se puede extraer para los otros algoritmos estudiados.

Gracias a los requerimientos que deben cumplir, los algoritmos enfocados a streams poseen un gran potencial para poder ser usados en aplicaciones que necesiten capacidad de procesamiento en tiempo real, cuyo número aumenta a medida que se desarrolla la tecnología. Algunos campos como la robótica, la detección de intrusiones en infraestructuras críticas o el control de procesos podrían resultar particularmente beneficiados.

Dicho potencial ha causado que se desarrolle una amplia cantidad de literatura dedicada al análisis y propuesta de nuevos algoritmos con mejores prestaciones; sin embargo, aún existen cuestiones pendientes. Una de las más importantes es la de la adaptación al cambio, representado por la deriva conceptual. En las pruebas realizadas, el cambio del parámetro C en la DELM producía cambios en la evolución de la precisión en el tiempo (particularmente notable en la Figura 3.12), lo que se traduce en que el detector de cambio, DDM, alerta de la ocurrencia de deriva conceptual en distintos momentos en el mismo set de datos. Aunque este problema puede deberse únicamente a una falta de estabilidad de la DELM, se pone de manifiesto que los sistemas de detección de deriva basados en la evolución del error de clasificación pueden ser propensos a imprecisiones como las falsas alarmas en el caso anterior o a ignorar completamente la ocurrencia de cambio como sucede en la Figura 3.17. Que la detección del cambio esté supeditada al clasificador no resulta adecuado. Lograr una detección eficaz de los distintos tipos de deriva conceptual es uno de los principales retos a los que se enfrenta este tipo de sistemas y uno de los campos en los que más se centra la literatura.

A. Detalles de implementación

En este apéndice se detallan las características del software usado para la implementación del sistema de clasificación seleccionado. Los archivos para ejecutar son comparacion.py (A.6) y comparacionELM.py (A.7). El resto simplemente definen clases o funciones.

A.1. lector.py

Esta clase se encarga de leer los archivos de datos por bloques y se apoya sobre la biblioteca pandas. Solo acepta archivos con datos numéricos en los que la etiqueta de clase se sitúa en la última posición. Los parámetros del constructor son:

- *archivo*: Una cadena que especifica la localización del archivo de datos.
- *separador*: Una cadena que indica el separador entre los atributos de cada instancia en el archivo.
- *chunkSize*: Es un entero que define el número de instancias que lee en cada bloque. El valor por defecto es 1000.

Los métodos de la clase son:

- *getBloque(self)*: Devuelve los atributos y la etiqueta de clase de un número de instancias igual a *chunkSize*, procedentes de *archivo*, en forma de 2 arrays bidimensionales.
- *leerTodo(self)*: Igual que la operación anterior, pero en este caso se leen todas las instancias restantes del archivo y no solo *chunkSize*.

A.2. normalizador.py

Esta clase se encarga de realizar la estandarización de los datos numéricos y la codificación de atributos categóricos y etiquetas.

Los atributos de entrada numéricos se ven sometidos a un proceso de normalización definido por $X' = \frac{X-\mu}{\sigma}$, donde μ es la media y σ la desviación típica.

Los atributos de tipo categórico se codifican por el procedimiento one-hot, que transforma un número en un secuencia binaria con longitud igual al número de valores distintos y con ceros en todas las posiciones excepto en la correspondiente al valor original. Por ejemplo, si existen 3 valores posibles y se codifica un 2 el resultado sería 0 1 0.

Los parámetros del constructor son:

- *numericos*: Un array de números que indica la posición de los atributos numéricos a estandarizar. Por defecto es un array vacío.
- *categoricos:* Un array de números que indica la posición de los atributos de tipo categórico a codificar. Por defecto es un array vacío.
- *numCategoricos*: Un array de números de la misma longitud que *categoricos*. Cada número indica la cantidad de diferentes valores posibles que puede tomar el atributo designado en *categoricos*. Por defecto es un array vacío.
- *inicioCategoricos*: Es un entero que indica cual es el primer valor de los atributos de tipo categórico. Por defecto es 0.

Los métodos de la clase son:

- *normalizarNumericos(self, datos)*: *datos* es un array bidimensional que contiene las instancias. Este método realiza una estandarización de los atributos indicados en *numericos* y devuelve un array bidimensional de las mismas dimensiones de la entrada con la transformación realizada.
- *codificarCategoricos(self, datos): datos* es un array bidimensional que contiene las instancias. Este método realiza una codificación one-hot de los atributos indicados en *categoricos* y devuelve un array bidimensional con la transformación realizada. Debido a la naturaleza de la codificación, se incrementará la dimensionalidad de las instancias.
- *normalizar(self, datos)*: Realiza una llamada a *normalizarNumericos* y *codificarCategoricos* usando *datos* como argumento y devuelve un array bidimensional con los datos transformados. Si *numericos* y *categoricos* son vacíos no realiza la llamada correspondiente.
- *codificarClases(self, etiquetas, numeroClases, inicioEtiquetas)*: *etiquetas* es un array bidimensional que contiene las etiquetas de las instancias; *numeroClases* es un entero que indica el número de clases e *inicioEtiquetas* es un entero que define el número de clases distintas; por defecto es 0. Este método realiza una codificación one-hot de las etiquetas pasadas y devuelve un array bidimensional con las etiquetas transformadas.

A.3. capa.py

Esta clase define las capas que forman la ELM y realiza el grueso de las operaciones numéricas necesarias. Más información sobre los parámetros y funcionamiento de la ELM se puede obtener en el epígrafe 2.5.1. Los parámetros del constructor son:

- *numeroNodos*: Un entero que determina el número de neuronas de la capa.
- *dimensiónEntrada*: Un entero que indica la dimensionalidad (número de atributos) de los datos de entrada de la capa.
- *activacion*: Una cadena que indica la función de activación de las neuronas de la capa. Los valores admitidos son "*sigmoid*", "*tanh*", "*relu*" y "*softmax*".
- *penalizacion*: Un número que determina valor del parámetro *C* de la ELM que se aplica a la capa.

Los métodos de la clase son:

- *getResultado(self, datos)*: *datos* es un array bidimensional con la entrada de la capa. Esta función calcula y devuelve la matriz H de la capa.
- *getSalida(self, datos): datos* es un array bidimensional con la entrada de la capa. Esta función calcula y devuelve la salida de la capa: HB.
- *entrenar(self, datos, etiquetas)*: *datos* es un array bidimensional con la entrada de la capa. *etiquetas* es un array bidimensional que contiene las etiquetas correspondientes a *datos*. Esta función inicializa la capa y es necesario llamarla antes de realizar otras operaciones. Devuelve la salida de la capa al realizar el entrenamiento.
- *procesar(self, datos)*: simplemente realiza una llamada a *getSalida* con *datos* como argumento.
- *actualizar(self, datos, etiquetas): datos* es un array bidimensional con la entrada de la capa. *etiquetas* es un array bidimensional que contiene las

etiquetas correspondientes a *datos*. Esta función actualiza los parámetros de la capa.
- *setNumNodos(self, num): num* es un entero que indica el nuevo número de neuronas de la capa. Esta función permite cambiar el número de neuronas de la capa.
- *reiniciar(self)*: esta función reinicia el estado de la capa, de forma que es necesario volver a entrenarla para usarse.
- *sigmoide(self, datos): datos* es un array bidimensional con la entrada de la capa. Este método aplica la función sigmoide a *datos*.
- *tanh(self, datos): datos* es un array bidimensional con la entrada de la capa. Este método aplica la función tangente hiperbólica a *datos*.
- *relu(self, datos): datos* es un array bidimensional con la entrada de la capa. Este método aplica la función rectificadora lineal a *datos*.
- *softmax(self, datos): datos* es un array bidimensional con la entrada de la capa. Este método aplica la función softmax a *datos*.
- *def entrenarAcumulado(self, datos):* Esta función es la misma que *entrenar*, pero guarda resultados intermedios para su posterior uso.
- *def getSalidaAcumulado(self, datos)*: Esta función es la misma que *getSalida*, pero hace uso de resultados intermedios guardados para reducir el tiempo de cómputo.
- *procesarAcumulado(self, datos):* Esta función es la misma que *procesar*, pero guarda resultados intermedios para su posterior uso.
- *actualizarAcumulado(self, datos, etiquetas)*: Esta función es la misma que *actualizar*, pero hace uso de resultados intermedios guardados para reducir el tiempo de cómputo. Para usarse es necesario haber ejecutado *procesarAcumulado* previamente.

A.4. elm.py

Esta clase define la ELM y organiza su funcionamiento. Más información sobre los parámetros y funcionamiento de la ELM se puede obtener en el epígrafe 2.5.1. Los parámetros del constructor son:
- *numClases*: Un entero que especifica el número de clases de los datos con los que se va a trabajar.
- *numCapas*: Un entero que define el número de capas de la ELM. Por defecto es 2.
- *activacion*: Un array de cadenas que contiene las funciones de activación deseadas para cada capa. Los valores válidos se definen en la sección A.3. Por defecto es [“sigmoid”, “sigmoid”].
- *umbral*: Un número que determina el valor de la constante τ. Por defecto es 1.
- *constante*: Un entero que determina el valor de la constante c. Por defecto es 5.
- *penalizacion*: Un número que determina el valor de la constante C. Por defecto es 0.003.

Los métodos de la clase son:
- *inicializar(self, datos, etiquetas)*: *datos* es un array bidimensional con los ejemplos de entrenamiento. *etiquetas* es un array bidimensional que

contiene las etiquetas correspondientes a *datos*. Esta función inicializa la ELM. Es necesario ejecutar *inicializar* antes de realizar otras operaciones.

- *entrenarCapas(self, datos, etiquetas)*: *datos* es un array bidimensional con los ejemplos de entrenamiento. *etiquetas* es un array bidimensional que contiene las etiquetas correspondientes a *datos*. Esta función inicializa las capas de la ELM.
- *procesar(self, datos, etiquetas)*: *datos* es un array bidimensional con los ejemplos a clasificar. *etiquetas* es un array bidimensional que contiene las etiquetas correspondientes a *datos*. Esta función realiza la clasificación de los datos y la actualización de las capas. Devuelve el error de clasificación para el bloque de datos procesado.
- *calcularError(self, prediccion, etiquetas)*: *prediccion* es un array bidimensional con las estimaciones de la ELM. *etiquetas* es un array bidimensional que contiene las etiquetas reales. Esta función calcula el error de clasificación para el bloque de datos.
- *actualizarError(self, error)*: Error es un número que indica el error de clasificación del último bloque de datos procesado. Esta función realiza el cálculo del error global y actualiza p_{min} y s_{min}.
- *actualizarEstado(self, prediccion, etiquetas)*: *prediccion* es un array bidimensional con las estimaciones de la ELM. *etiquetas* es un array bidimensional que contiene las etiquetas reales. Esta función llama a *calcularError*, *actualizarError* y determina el estado de detección de deriva conceptual.
- *actualizarCapas(self, datos, etiquetas)*: *datos* es un array bidimensional con los ejemplos a clasificar. *etiquetas* es un array bidimensional que contiene las etiquetas correspondientes a *datos*. Esta función realiza la actualización de los parámetros de las capas de la ELM.
- *actualizarNumNodos(self)*: Esta función incrementa el número de nodos de las capas de la ELM en función del estado de detección de deriva conceptual.
- *actualizarELM(self, datos, etiquetas)*: *datos* es un array bidimensional con los ejemplos a clasificar. *etiquetas* es un array bidimensional que contiene las etiquetas correspondientes a *datos*. Esta función llama a *actualizarCapas*, *actualizarNumNodos* y *entrenarCapas* en función del estado de detección de deriva conceptual.

A.5. evaluacion.py

En este archivo se definen varias funciones usadas para realizar la evaluación de la ELM. Las funciones son:

- *evaluarELM(archivo, numClases, bloque, separador, numericos, categoricos, numCategoricos, inicioCategoricos, inicioEtiquetas, penalizacion)*: Esta función crea una ELM de 2 capas, junto a una instancia de Lector y Normalizador como apoyo, y la evalúa, obteniendo como salida 4 arrays que contienen la siguiente información: error total, error por bloque de datos, consumo de memoria y tiempo de ejecución. Todos los arrays tienen una entrada por cada bloque de datos procesado. Los argumentos de esta función corresponden a los constructores de las clases creadas: *numericos, categoricos, numCategoricos, inicioCategoricos* e *inicioEtiquetas* a Normalizador (sección A.2); *archivo, bloque* y *separador*

a Lector (sección A.1); *numClases* y *penalizacion* a ELM (sección A.4). Por defecto, separador es una coma; *numericos, categoricos* y *numCategoricos* son arrays vacíos; *inicioCategoricos* e *inicioEtiquetas* tienen valor 1 y *penalizacion* es 0.003.

- *imprimir(titulo, errorTotal, tiempo, memoria)*: Esta función imprime varios datos de interés respecto al rendimiento: la precisión global, el tiempo de ejecución total y el tiempo de ejecución medio por bloque de datos y el consumo de memoria medio. *errorTotal*, *tiempo* y *memoria* son los arrays de datos con la información adecuada (los obtenidos al ejecutar *evaluarELM* en la práctica). *titulo* es una cadena para encabezar la información anterior.
- *plotComparacion(datos, ejeX, leyendas, labelx, labely, miny, maxy, titulo, guardar, archivo)*: Esta función se encarga de generar las gráficas de datos. *datos* es un array que contiene varios arrays de datos que serán representados en la misma gráfica. *ejeX* es un array numérico que establece la escala de valores representada en el eje X y con la misma longitud que los arrays de *datos*. *labelx* y *labely* son 2 cadenas que determinan las etiquetas que se mostrarán en los respectivos ejes. *miny* y *maxy* son números que marcan, respectivamente, el mínimo y máximo valor del eje Y; por defecto tienen el valor None, por lo que se determinan automáticamente en función de los datos. *titulo* es una cadena que definirá el título de la gráfica; por defecto es la cadena vacía. *guardar* es un valor booleano que determina si se guardará la gráfica como un archivo png; por defecto tiene valor *False*. *archivo* es una cadena que indica donde se guardará la gráfica; solo tiene efecto si se le ha asignado un valor verdadero a *guardar*.

A.6. comparacion.py

Al ejecutar este archivo se realiza la comparación entre la ELM y el MLP. Las salidas se obtienen usando funciones del archivo evaluacion.py (sección A.5). Se generan gráficas con la función *plotComparacion* y se imprime por pantalla con *imprimir*. La evaluación del MLP se lleva a cabo mediante la siguiente función:

- *def evaluarMLP(archivo, numClases, bloque, separador, numericos, categoricos, numCategoricos, inicioCategoricos, inicioEtiquetas)*: Los argumentos, sus valores por defecto y la salida de esta función son iguales a las de *evaluarELM*.

Los valores usados tanto para la ELM como para el MLP se pueden ajustar cambiando las constantes correspondientes al principio del archivo.

A.7. comparacionELM.py

Al ejecutar este archivo se comparan varias ELM con diferentes parámetros. Las salidas se son las mismas que en comparacion.py. Se definen una serie de constantes al inicio del archivo para establecer los valores comunes, pero los parámetros diferentes es necesario establecerlos en las llamadas a *evaluarELM*.

B. Generación de los sets de datos

Los sets de datos sintéticos se han generado usando la plataforma MOA. A continuación se describen los comandos necesarios para obtenerlos y su significado.

MOA 16.04 requiere de Java 6 SDK o superior para funcionar correctamente y es compatible tanto con Windows, MAC y Linux. Para empezar a trabajar simplemente hay que descomprimir el archivo correspondiente, descargable desde [9].

El comando usado desde una terminal para realizar la escritura de un stream a un archivo es el siguiente:

java -cp <ruta de moa.jar> -javaagent:<ruta de sizeofag.jar> moa.DoTask "WriteStreamToARFFFile -s <stream deseado> -f <ruta de archivo de salida> -m <número de instancias deseadas> -h"

La opción *-s* indica el generador de stream a usar; *-f* el archivo de salida; *-m* el número de instancias a generar y *-h* simplemente suprime el encabezado del archivo de salida y puede omitirse si se quiere.

Para poder especificar las opciones propias a cada stream de datos es necesario introducirlo entre paréntesis. Por ejemplo:

java -cp <ruta de moa.jar> -javaagent:<ruta de sizeofag.jar> moa.DoTask "WriteStreamToARFFFile -s (generators.WaveformGenerator -n) -f datos.txt -m 1000000 -h"

Los streams generados y las opciones usadas se describen a continuación. Información más detallada puede encontrarse en el manual de MOA [10]. Para todos los generadores se crearon 1000000 instancias.

Hyperplane. El generador es el siguiente:

generators.HyperplaneGenerator -c 2-a 10 -k 10 -t 0.001 -n 5

-c indica el número de clases; *-a* el número de atributos; *-k* el número de atributos en los que se introduce deriva; *-t* la magnitud de la deriva para cada ejemplo y *-n* el porcentaje de ruido.

LED. El generador es el siguiente:

generators.LEDGeneratorDrift -n 10 -s -d 7

-n indica la probabilidad de que los atributos estén invertidos; *-s* elimina atributos irrelevantes, dejando solo 7 y *-d* indica el número de atributos en los que se introduce deriva.

RandomRBF. El generador es el siguiente:

generators.RandomRBFGeneratorDrift -c 5 -a 10 -n 50 -s 0.00001 -k 50

-c determina el número de clases; *-a* el número de atributos; *-n* el número de centroides; *-s* la velocidad de desplazamiento de los centroides (para introducir deriva) y *-k* el número de centroides con deriva.

SEA. El generador es el siguiente:

generators.SEAGenerator -n 10

-*n* indica el porcentaje de ruido introducido.

Lista de referencias

[1] C. C. Aggarwal, *Data classification : algorithms and applications*. 2015.

[2] C. Aggarwal, J. Han, J. Wang, and P. Yu, "On demand classification of data streams," *Proc. ACM KDD Conf.*, pp. 503–508, 2004.

[3] C. Aggarwal and P. Yu, "LOCUST: An Online Analytical Processing Framework for High Dimensional Classification of Data Streams," *Data Eng. 2008. ICDE 2008. IEEE 24th Int. Conf.*, pp. 426–435, 2008.

[4] A. Bifet, R. Gavalda, and R. Gavaldà, "Learning from Time-Changing Data with Adaptive Windowing.," *Sdm*, vol. 7, p. 2007, 2007.

[5] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA Massive Online Analysis," *J. Mach. Learn. Res.*, vol. 11, pp. 1601–1604, 2011.

[6] A. Bifet, G. Holmes, B. Pfahringer, and E. Frank, "Fast perceptron decision tree learning from evolving data streams," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6119 LNAI, no. PART 2, pp. 299–310, 2010.

[7] A. Bifet and R. Kirkby, *Data Stream Mining - A Practical Approach*, vol. 8, no. May. 2009.

[8] A. Bifet and R. Kirkby, "MOA Datasets," 2009. [Online]. Available: http://moa.cms.waikato.ac.nz/datasets/. [Accessed: 17-May-2017].

[9] A. Bifet and R. Kirkby, "Descarga de MOA," 2016. [Online]. Available: http://moa.cms.waikato.ac.nz/downloads/. [Accessed: 15-May-2017].

[10] A. Bifet, R. Kirkby, P. Kranen, and P. Reutemann, "Massive online analysis manual," *Univ. Waikato, New Zeal. Cent. Open Softw. Innov.*, no. March, 2009.

[11] A. Bifet, J. Read, I. Žliobaite, B. Pfahringer, and G. Holmes, "Pitfalls in Benchmarking Data Stream Classification and How to Avoid Them," vol. 8188, pp. 465–479, 2013.

[12] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. 1984.

[13] D. Brzeziński, "Mining Data Streams With Concept Drift," *Cs.Put.Poznan.Pl*, p. 89, 2010.

[14] D. Brzeziński and J. Stefanowski, "Accuracy updated ensemble for data streams with concept drift," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6679 LNAI, no. PART 2, pp. 155–163, 2011.

[15] J. Catlett, "Statlog (Shuttle) Data Set." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Statlog+(Shuttle). [Accessed: 15-May-2017].

[16] Charu C. Aggarwal, "A Survey on Stream Classification Algorithm," *Data Classif. Algorithms Appl.*, pp. 245–273, 2014.

[17] F. Chollet and others, "Keras," 2015. [Online]. Available: https://github.com/fchollet/keras.

[18] M. Deckert, “Incremental rule-based learners for handling concept drift: An overview,” *Found. Comput. Decis. Sci.*, vol. 38, no. 1, 2013.

[19] C. Domeniconi and D. Gunopulos, “Incremental support vector machine construction,” *Proc. 2001 IEEE Int. Conf. Data Min.*, pp. 589–592, 2001.

[20] P. Domingos and G. Hulten, “Mining High-Speed Data Streams,” *Proc. Sixth ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, pp. 71–80, 2000.

[21] J. Gama, *Knowledge Discovery from Data Streams*, 1st ed. Chapman & Hall/CRC, 2010.

[22] J. Gama, P. Medas, G. Castillo, and P. Rodrigues, “Learning with drift detection,” *Brazilian Symp. Artif. Intell.*, pp. 286–295, 2004.

[23] J. Gama, R. Sebastião, and P. P. Rodrigues, “On evaluating stream learning algorithms,” *Mach. Learn.*, vol. 90, no. 3, pp. 317–346, 2013.

[24] J. Gama, R. Sebastião, and P. P. Rodrigues, “Issues in evaluation of stream learning algorithms,” *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discov. data Min. - KDD ’09*, pp. 329–337, 2009.

[25] J. Gama, I. Žliobait, A. Bifet, M. Pechenizkiy, and A. Bouchachia, “A Survey on Concept Drift Adaptation,” *ACM Comput. Surv.*, vol. 46, no. 4, p. 44:1--44:37, 2014.

[26] Guang-Bin Huang, Hongming Zhou, Xiaojian Ding, and Rui Zhang, “Extreme Learning Machine for Regression and Multiclass Classification,” *IEEE Trans. Syst. Man, Cybern. Part B*, vol. 42, no. 2, pp. 513–529, 2012.

[27] L. K. Hansen and P. Salamon, “Neural Network Ensembles,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, no. 10, pp. 993–1001, 1990.

[28] M. Harries, U. Nsw-cse-tr, and N. S. Wales, “SPLICE-2 Comparative Evaluation: Electricity Pricing,” 1999.

[29] G. Hulten, “Mining Time-Changing Data Streams,” pp. 97–106, 2001.

[30] J. D. Hunter, “Matplotlib: A 2D graphics environment,” *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, 2007.

[31] D. P. Kingma and J. L. Ba, “Adam: a Method for Stochastic Optimization,” *Int. Conf. Learn. Represent. 2015*, pp. 1–15, 2015.

[32] R. Klinkenberg and T. Joachims, “Detecting concept drift with support vector machines,” *Proc. Seventeenth Int. Conf. Mach. Learn.*, vol. 11, no. May 2000, pp. 487–494, 2000.

[33] Y.-N. Law and C. Zaniolo, “An Adaptive Nearest Neighbor Classification Algorithm for Data Streams,” in *Knowledge Discovery in Databases: PKDD 2005*, 2005, pp. 108–120.

[34] M. Lazarescu, S. Venkatesh, and H. Bui, “Using multiple windows to track concept drift,” *Intell. Data Anal.*, pp. 1–28, 2004.

[35] W. McKinney and P. D. Team, “Pandas - Powerful Python Data Analysis Toolkit,” *Pandas - Powerful Python Data Anal. Toolkit*, p. 1625, 2015.

[36] V. Nathan, “Accurate Streaming Support Vector Machines,” 2014.

[37] NumPy Community, “NumPy Reference,” 2017.

[38] E. Page, “Continuous inspection schemes,” *Biometrika*, vol. 41, no. 1, pp. 100–115, 1954.

[39] P. Rai, H. Daumé, and S. Venkatasubramanian, “Streamed learning: One-pass SVMs,” *IJCAI Int. Jt. Conf. Artif. Intell.*, pp. 1211–1216, 2009.

[40] G. Rodola, “psutil,” 2008. [Online]. Available: https://github.com/giampaolo/psutil.

[41] G. van Rossum and Python development team, “The Python Language Reference Manual,” 2017.

[42] R. Sebastião and J. Gama, “A study on change detection methods,” *14th Port. Conf. Artif. Intell.*, pp. 353–364, 2009.

[43] J. Stefanowski and D. Brzezinski, “Stream Classification,” in *Encyclopedia of Machine Learning and Data Mining*, 2016.

[44] W. N. Street and Y. Kim, “A streaming ensemble algorithm (SEA) for large-scale classification,” in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’01*, 2001, pp. 377–382.

[45] N. A. Syed, H. Liu, and K. K. Sung, “Handling Concept Drifts in Incremental Learning with Support Vector Machines,” *Proc. Fifth ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, pp. 317–321, 1999.

[46] H. Wang, W. Fan, P. S. Yu, and J. Han, “Mining concept-drifting data streams using ensemble classifiers,” *Ninth ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, vol. 2, no. 1, pp. 226--235, 2003.

[47] S. Xu and J. Wang, “Dynamic Extreme Learning Machine for Data Stream Classification,” *Neurocomputing*, 2017.

[48] I. Zliobaite, “How good is the Electricity benchmark for evaluating concept drift adaptation,” *arXiv Prepr. arXiv1301.3524*, pp. 1–6, 2013.

[49] I. Žliobaite, “Controlled permutations for testing adaptive learning models,” *Knowl. Inf. Syst.*, vol. 39, no. 3, pp. 565–578, 2014.

[50] I. Žliobaite, “Learning under Concept Drift: an Overview,” *Training*, vol. abs/1010.4, pp. 1–36, 2010.

[51] I. Žliobaitė, A. Bifet, J. Read, B. Pfahringer, and G. Holmes, “Evaluation methods and decision theory for classification of streaming data with temporal dependence,” *Mach. Learn.*, vol. 98, no. 3, pp. 455–482, 2014.