Stream Learning

Modelos Avanzados de Aprendizaje Automático I

El *Stream Learning*, también conocido como aprendizaje en flujo o aprendizaje continuo, es una rama del Aprendizaje Automático que se enfoca en el procesamiento y análisis de datos que llegan de manera continua y en tiempo real. A diferencia de los métodos tradicionales de aprendizaje automático que se basan en conjuntos de datos estáticos y finitos, el *Stream Learning* trabaja con flujos de datos que pueden ser potencialmente infinitos y en constante cambio.

Como se ve en teoría, algunas características claves del *Stream Learning* son las siguientes:

- Procesamiento incremental. El modelo de aprendizaje se actualiza de manera continua a medida que llegan nuevos datos. Esto permite al modelo adaptarse a cambios en los datos a lo largo del tiempo, lo cual es crucial en entornos dinámicos.
- Memoria limitada. Dado que los datos pueden ser infinitos, no es factible almacenar todo el historial de datos. Los algoritmos de *Stream Learning* deben ser capaces de aprender utilizando una cantidad limitada de memoria.
- Tiempo de respuesta rápido. Es esencial que los modelos de *Stream Learning* puedan procesar y analizar los datos en tiempo real o casi en tiempo real, para proporcionar predicciones y adaptaciones rápidas.
- Deriva de concepto (*Concept Drift*): Los datos en flujo pueden cambiar sus características estadísticas con el tiempo, lo que se conoce como "cambio de concepto" o "deriva de concepto". Los algoritmos deben detectar y adaptarse a estos cambios para mantener la precisión del modelo.

Este ejercicio, por lo tanto, está muy relacionado con la práctica anterior sobre aprendizaje incremental. De hecho, se hará uso de las funciones desarrolladas en ella.

En este ejercicio se utilizará un conocido *dataset* de *Stream Learning*, de nombre *Electricity*, cuya descripción está en el ejercicio 1, así como la función para cargarlo y preprocesarlo. Además, este *dataset* es conocido también por tener una clara deriva del concepto.

En este ejercicio se utilizará SVM y kNN para clasificar los datos de este *dataset*. Con respecto a SVM, se hará la comparación de los resultados obtenidos con el modelo ISVM desarrollado en el ejercicio anterior, con el SVM clásico que utiliza únicamente una ventana deslizante como memoria. Estos modelos de SVM se compararán con el uso de un kNN con una ventana deslizante como memoria.

Para realizar estas comparativas, se hará una simulación de la recepción de datos a través de un flujo. Aunque un escenario habitual en *Stream Learning* es procesar los datos entrantes uno a uno (es decir, actualizar el modelo con cada dato nuevo), en general no tendría por qué ser así. De hecho, existen muchos escenarios en los que esta opción es impracticable, dada la alta frecuencia de los datos. Por ejemplo, en señales acústicas se toman muchos datos cada segundo, por lo que no se podría actualizar el modelo a medida que llega cada muestra. Por lo tanto, en general, los modelos se actualizarán para cada lote de datos nuevo, que podría constar de un único dato, pero, para que el desarrollo de estas funciones no sea demasiado lento, no será de esta manera. Por cada lote nuevo de datos, antes de actualizar el modelo, este se prueba con los datos nuevos, para averiguar el rendimiento del modelo actual. Una vez que se ha probado, se actualiza con estos datos nuevos.

Además, este tipo de modelos habitualmente tienen una memoria, que constituye una ventana temporal con una estructura de tipo FIFO (*First In, First Out*) donde se van almacenando los datos que van llegando, descartando los más antiguos. Como se ha dicho en el ejercicio anterior, se seguirá el criterio de que en un vector o matriz, los primeros elementos serán los más antiguos, y los últimos serán los más recientes. Para garantizar la repetibilidad de los resultados, este criterio debe seguirse a la hora de trabajar con los datos de este ejercicio.

En este ejercicio, se pide realizar las siguientes funciones:

- *initializeStreamLearningData*. Esta función inicializa los datos para realizar la simulación de *Stream Learning*: carga de datos y división de estos en lotes que van llegando a través del flujo de datos. Esta función recibe como parámetros:
    - *datasetFolder*, de tipo *String*, con el nombre de la carpeta donde está el *dataset* de *Stream Learning* a cargar.
    - *windowSize*, de tipo *Int*, con el tamaño de la ventana (en número de instancias) que tendrá la memoria inicial.
    - *batchSize*, de tipo *Int*, con el tamaño de los lotes de datos para actualizar el modelo.

  Esta función debe devolver una tupla donde el primer elemento es de tipo *Batch* y contiene la memoria, y el segundo es un vector donde cada elemento es de tipo *Batch*, es decir, son los lotes que datos que llegan por el flujo de datos. Esto permitirá hacer una simulación de la llegada de *batches* a través del flujo de datos.

  Para hacer esto, el primer paso es cargar los datos mediante la función *loadStreamLearningDataset* desarrollada en el primer ejercicio, que devuelve un lote de datos, es decir, la tupla que devuelve se puede tomar como si fuese de tipo *Batch*. De este

lote de datos, se toman las primeras *windowSize* instancias para constituir la memoria (mediante la función *selectInstances* del ejercicio anterior), y el resto (nuevamente con *selectInstances*) se usarán para realizar una llamada a la función *divideBatches* del ejercicio anterior, indicando como parámetro opcional *shuffleRows=false*, para que no desordene las muestras (estas llegan de forma ordenada en el tiempo).

Al crear la memoria de esta manera, las primeras muestras de la misma la formarán los datos más antiguos, y las últimas serán los datos más recientes.

Para el desarrollo de esta función no se permite el uso de bucles.

- *addBatch!* Esta función recibe una memoria y un lote de datos recién llegados, y actualiza la memoria con los datos nuevos. Recibe los siguientes parámetros:
    - *memory*, de tipo *Batch*, con la memoria actual.
    - *newBatch*, de tipo *Batch*, con el nuevo lote de datos.

  Esta función actualizará la memoria con el nuevo lote de datos. Para ello, la primera acción que debe realizar es "desplazar" los datos que haya en la memoria hasta el principio, en una cantidad igual al número de datos en el lote, descartando los primeros por ser los más antiguos. Esto se debe realizar tanto en la matriz de entradas como en el vector de salidas deseadas. Una vez esto ha sido realizado, se procede a copiar los datos del nuevo lote al final de la memoria, sin modificar el orden. Todo esto se hará en la propia memoria pasada como primer argumento a esta función, sin crear una copia de la misma, puesto que no se devolverá una nueva memoria, sino que se modificará esta.

  Para el desarrollo de esta función no se permite el uso de bucles.
- *streamLearning_SVM*. Realiza una simulación de un modelo SVM en un entorno con un flujo de datos. Recibe como parámetros:
    - *datasetFolder*, de tipo *String*, con la carpeta donde está el archivo de datos.
    - *windowSize*, de tipo *Int*, con el tamaño de la ventana de la memoria.
    - *batchSize*, de tipo *Int*, con el tamaño de cada lote de datos.
    - *kernel*, de tipo *String*, con el *kernel* a usar en el entrenamiento.
    - *C*, de tipo *Real*, con el valor de C a usar en cada entrenamiento.

  Además, esta función recibe como parámetros opcionales:
    - *degree*, de tipo *Real*, *gamma*, de tipo *Real*, y *coef0*, de tipo *Real*. Contienen los valores de los hiperparámetros de cada tipo de *kernel*.

  En este caso, con la llegada de cada nuevo lote de datos, se hará test con ese lote, se actualizará la memoria y el SVM se entrenará con los datos que se tenga en la memoria. La función deberá devolver un vector con las precisiones en test de los modelos obtenidos antes de la actualización de la memoria con cada *batch*. Es decir, este vector deberá tener

una longitud igual al número de *batches*, donde el primer elemento es la precisión con el modelo entrenado únicamente con la memoria inicial y evaluado en el primer *batch*, el segundo elemento es la precisión con el modelo obtenido al actualizar la memoria con el primer *batch* y evaluado en el segundo *batch*, etc. En este esquema, no se tiene una precisión después de actualizar la memoria con el último *batch*, esto viene provocado por el hecho de que no hay un *batch* siguiente sobre el que hacer test.

Para realizar esto, la función deberá hacer los siguientes pasos:

- Inicializar memoria y *batches* mediante la función *initializeStreamLearningData*.
- Entrenar el primer SVM mediante la función *trainSVM* de la práctica anterior.
- Crear un vector con tantos elementos como lotes de datos, para almacenar las precisiones.
- Desarrollar un bucle, con tantos ciclos como *batches*. En el ciclo i, hacer:
  - Hacer test del modelo actual (función *predict*) con el i-ésimo *batch*, calcular la precisión y almacenarla en el vector.
  - Actualizar la memoria con el i-ésimo *batch* mediante la función *addBatch!*
  - Entrenar un nuevo SVM con la memoria actualizada que se tiene. Este será el nuevo modelo.
- Finalmente, devolver el vector con las precisiones usando cada *batch* como test.

Para el desarrollo de esta función se permite el uso de un único bucle, que itere sobre el vector de *batches*.

- *streamLearning_ISVM*. Realiza una simulación de un modelo ISVM en un entorno con un flujo de datos. Recibe como parámetros:
  - *datasetFolder*, de tipo *String*, con la carpeta donde está el archivo de datos.
  - *windowSize*, de tipo *Int*, con el tamaño de la ventana de la memoria.
  - *batchSize*, de tipo *Int*, con el tamaño de cada lote de datos.
  - *kernel*, de tipo *String*, con el *kernel* a usar en el entrenamiento.
  - *C*, de tipo *Real*, con el valor de C a usar en cada entrenamiento.

  Además, esta función recibe como parámetros opcionales:
  - *degree*, de tipo *Real*, *gamma*, de tipo *Real*, y *coef0*, de tipo *Real*. Contienen los valores de los hiperparámetros de cada tipo de *kernel*.

  Esta función se diferencia del anterior en que se entrenará un SVM siguiendo la filosofía de aprendizaje incremental, es decir, con la llegada de cada nuevo lote de datos se entrenará un SVM de forma incremental, y no se tendrá una memoria. En lugar de tener una memoria, los vectores de soporte como resultado del anterior entrenamiento se utilizarán para realizar también el siguiente entrenamiento, descartando aquellos que sean demasiado

antiguos. Por lo tanto, se deberá llevar la cuenta de la antigüedad de cada vector de soporte. Al igual que antes, la función deberá devolver un vector con las precisiones en test de cada *batch*, pero la secuencia de pasos es un poco distinta, puesto que será necesario tener en cuenta los vectores de soporte de cada entrenamiento. Los pasos a llevar a cabo son:

- Inicializar memoria y *batches* mediante la función *initializeStreamLearningData*. En este caso no hay memoria; sin embargo, esta función se puede utilizar para separar el primer *batch* para hacer el entrenamiento inicial (*warm start*), sin vectores de soporte que aportar. Para ello, se puede llamar a *initializeStreamLearningData* indicando como segundo y tercer argumento el mismo valor, igual a *batchSize*. De esta manera, esta función devuelve una tupla con dos valores, por este orden:
  - *Batch* inicial, utilizado para entrenar el SVM con un conjunto inicial de datos, para hacer un *warm start*.
  - Flujo de datos, representado como un vector de *batches*, que se utilizará para entrenar el ISVM en un bucle con tantas iteraciones como *batches* en este vector.
- Entrenar el primer SVM mediante la función *trainSVM* del ejercicio anterior con el *batch* inicial (*warm start*). Como resultado, el segundo valor de la tupla de salida serán los vectores de soporte. Como tercer valor, se tiene una tupla en la que el segundo elemento serán los índices de las instancias utilizadas como vectores de soporte, llamado *indicesSupportVectorsInFirstBatch*.
- Crear un vector con la edad de los patrones, que, siguiendo el criterio de que los valores más antiguos serán los primeros, será igual a *batchSize* para el primero de la matriz (el más antiguo), y 1 para el último (el más reciente). A partir de este vector, se puede tomar el vector de edad de los vectores de soporte referenciando los elementos en las posiciones dadas por *indicesSupportVectorsInFirstBatch*.
- Crear un vector con tantos elementos como lotes de datos, para almacenar las precisiones.
- Desarrollar un bucle, con tantos ciclos como *batches* del flujo de datos, es decir, tantos ciclos como elementos del vector de *batches*. En el ciclo i, hacer:
  - Hacer test del modelo actual con el i-ésimo *batch*, calcular la precisión y almacenarla en el vector.
  - Actualizar el vector de edad de los vectores de soporte sumando a su

edad el número de elementos del nuevo lote de datos. Es importante darse cuenta de que la edad que se suma no es igual a *batchSize*, pasado como argumento a esta función, puesto que el último lote de datos tendrá un número de instancias inferior a este valor.

- Seleccionar, del lote de datos con los vectores de soporte, aquellos cuya edad es inferior o igual a *windowSize* (función *selectInstances* del ejercicio anterior). Será necesario seleccionar igualmente los elementos del vector de edad para que ambas estructuras (vectores de soporte y edades) sean coherentes en el número de instancias.
- Entrenar un nuevo SVM utilizando la función desarrollada en el ejercicio anterior con el nuevo *batch*, y los vectores de soporte resultantes de la operación anterior que se tienen. Este será el nuevo modelo. La función de entrenamiento devolverá también, como tercer valor de una tupla, dos vectores de índices de los nuevos vectores de soporte, referidos a los anteriores vectores de soporte y al *batch* de este entrenamiento.
- Crear un nuevo lote de datos con los nuevos vectores de soporte, uniendo los siguientes conjuntos de datos, mediante la función *joinBatches*, por este orden, siguiendo el criterio de que los datos más antiguos estén al principio:
    - Del conjunto de vectores de soporte utilizado en la llamada, se toman las instancias correspondientes al primer vector devuelto en el tercer valor de la llamada a *trainSVM*, con los índices de los anteriores vectores de soporte seleccionados. Para esto, utilizar la función *selectInstances*.
    - Del *batch* utilizado, se toman las instancias correspondientes al segundo vector devuelto en el tercer valor de la llamada a *trainSVM*, que contiene los índices de las instancias del *batch* utilizadas como vectores de soporte. Para esto, utilizar la función *selectInstances*.
- Crear el vector de edades de los nuevos vectores de soporte de forma similar, concatenando estos dos vectores (función *vcat*), por este orden, siguiendo de nuevo el criterio de que los datos más antiguos estén al principio:
    - A partir del vector anterior de edades de los vectores de soporte anteriores, se toman las edades de estos utilizando, al igual que

antes, el primer vector devuelto en el tercer valor de la llamada a *trainSVM*, con los índices de los anteriores vectores de soporte seleccionados.

- Crear un vector descendente de valores de N a 1, donde N es el número de instancias de este *batch*. De este vector, seleccionar las utilizadas como vectores de soporte referenciándolas, al igual que antes, con el segundo vector devuelto en el tercer valor de la llamada a *trainSVM*.

  - Finalmente, devolver el vector con las precisiones.

Dada la complejidad de esta función, se recomienda que esta incluya una serie de líneas de *programación defensiva* para detectar errores en el desarrollo, que realicen comprobaciones como que el vector de edades tenga el mismo número de elementos que el *batch* de los vectores de soporte.

Para el desarrollo de esta función se permite el uso de un único bucle, que itere sobre el vector de *batches*.

- *euclideanDistances*. Función que, dado un *batch* de datos y una instancia, calcula las distancias euclídeas de esta instancia a todas las del *batch*. Recibe como parámetros:
  - *dataset*, de tipo *Batch*, con el conjunto de datos.
  - *instance*, de tipo *AbstractArray{<:Real,1}*, vector con la instancia.

  Esta función se puede desarrollar de forma vectorial. Para ello, es posible calcular la diferencia entre la instancia y cada una de las filas de la matriz de entradas del *batch*. Sin embargo, no es posible hacer un *broadcast* de la operación de diferencia, puesto que *instance* es un vector, y, por lo tanto, sería tratado como una columna. Por lo tanto, es necesario trasponerlo, tras lo cual ya se puede hacer el *broadcast* de la operación de diferencia. Una vez hecho, los elementos de la matriz resultante se pueden elevar al cuadrado (nuevo *broadcast*), y posteriormente calcular la suma a través de cada fila con la función *sum* y utilizando el *keyword dims=2*. Como resultado, dará una matriz de dimensiones Nx1, donde N es el número de instancias en la memoria, que contiene la suma de los cuadrados de las diferencias entre la instancia pasada como segundo argumento, y cada instancia de la memoria. Para calcular la distancia, simplemente es necesario hacer la raíz cuadrada de cada elemento de esta matriz (*broadcast* de la función *sqrt*), y devolver esta matriz con una columna como un vector (esto se puede hacer con la función *vec*).

  Para el desarrollo de esta función no se permite el uso de bucles.

- *nearestElements*. Función que, dado un *batch* de datos y una instancia, devuelve otro *batch* con las instancias más cercanas a la pasada como parámetro. Recibe como parámetros:

- *dataset*, de tipo *Batch*, con el conjunto de datos.
- *instance*, de tipo *AbstractArray{<:Real,1}*, vector con la instancia.
- *k*, de tipo *Int*, con el número de vecinos más cercanos a tomar.

Esta función se apoya en la función anterior *euclideanDistances* para calcular la distancia de la instancia al *batch* pasado como parámetro con la memoria. Una vez se tienen esas distancias, se toman los índices de las instancias con distancia más pequeña. Esto hará con la función *partialsortperm*. Finalmente, se tomarán las instancias del *batch* mediante la función *selectInstances* desarrollada previamente. El resultado a devolver deberá ser de tipo *Batch*.

Para el desarrollo de esta función no se permite el uso de bucles.

- *predictKNN*. Función que emite una predicción sobre una instancia utilizando kNN. Recibe como parámetros:
  - *dataset*, de tipo *Batch*, con el lote de datos de entrenamiento
  - *instance*, de tipo *AbstractArray{<:Real,1}*, con la instancia sobre la que realizar la predicción.
  - *k*, de tipo *Int*, con el número de vecinos.

  Esta función se apoya en la función anterior *nearestElements* para calcular las instancias más cercanas del *batch* a la instancia pasada como parámetro. De esas instancias, se toman las salidas deseadas mediante la función *batchTargets* desarrollada previamente, y se devuelve el valor más común, esto se hará mediante la función *mode* de *StatsBase*. El valor a devolver, por lo tanto, deberá tener el mismo tipo que las salidas deseadas del *batch*.

  Para el desarrollo de esta función no se permite el uso de bucles.
- *predictKNN*. Función sobrecargada con el mismo nombre que la anterior, realiza la misma operación con la salvedad de que, en lugar de recibir una única instancia, recibe un conjunto de instancias. Recibe como parámetros:
  - *dataset*, de tipo *Batch*, con el lote de datos de entrenamiento
  - *instances*, de tipo *AbstractArray{<:Real,2}*, con el conjunto de instancias sobre las que realizar las predicciones.
  - *k*, de tipo *Int*, con el número de vecinos.

  Esto se puede hará mediante una *comprehension* que itere sobre las filas de *instances* utilizando la función *eachrow*, y haciendo llamadas a la función anterior.

  Para el desarrollo de esta función no se permite el uso de bucles.
- *streamLearning_KNN*. Realiza una simulación de un entorno de *Stream Learning* con el *dataset* de este ejercicio, y las funciones implementadas de kNN. Recibe como parámetros:
  - *datasetFolder*, de tipo *String*, con la carpeta donde está el *dataset* de *Stream*

Learning.

- *windowSize*, de tipo *Int*, con el tamaño de la memoria en número de instancias.
- *batchSize*, de tipo *Int*, con el tamaño del *batch*.
- *k*, de tipo *Int*, con el número de vecinos a considerar por el algoritmo kNN.

Esta función es muy similar a la función *streamLearning_SVM*; de hecho, es un poco más sencilla que esa. Las únicas diferencias son que en esta no se entrena ningún modelo en ninguna parte, y que, en lugar de utilizar la función *predict*, se utiliza la función *predictKNN* desarrollada previamente.

Al igual que en la función *streamLearning_SVM*, esta función deberá devolver un vector con las precisiones usando cada *batch* como test.

Para el desarrollo de esta función se permite el uso de un único bucle, que itere sobre el vector de *batches*.

Una vez realizadas estas funciones, la parte de experimentación de este ejercicio es muy sencilla, y consiste en ejecutar las funciones de simulación de entornos de *Stream Learning* con SVM, ISVM y kNN con distintos tamaños de memoria y de *batch*, y comparar las precisiones obtenidas.

- ¿Qué diferencia hay entre SVM e ISVM cuando el tamaño del *batch* es igual al tamaño de la memoria? ¿A qué es debido esto?
- ¿Qué modelos ofrecen mejor precisión cuando el tamaño de la memoria y del *batch* es alto, y cuando estos tamaños son bajos? ¿A qué es debido esto?
- ¿En qué condiciones kNN mejora los resultados de SVM e ISVM?

Firmas de las funciones:

A continuación se muestran las firmas de las funciones a realizar en los ejercicios propuestos. Tened en cuenta que, dependiendo de cómo se defina la función, esta puede contener o no la palabra reservada *function* al principio:

```
function initializeStreamLearningData(datasetFolder::String, windowSize::Int, batchSize::Int)
function addBatch!(memory::Batch, newBatch::Batch)
function streamLearning_SVM(datasetFolder::String, windowSize::Int, batchSize::Int,
        kernel::String, C::Real; degree::Real=1, gamma::Real=2, coef0::Real=0.)
function streamLearning_ISVM(datasetFolder::String, windowSize::Int, batchSize::Int,
        kernel::String, C::Real; degree::Real=1, gamma::Real=2, coef0::Real=0.)
function euclideanDistances(dataset::Batch, instance::AbstractArray{<:Real,1})
function nearestElements(dataset::Batch, instance::AbstractArray{<:Real,1}, k::Int)
function predictKNN(dataset::Batch, instance::AbstractArray{<:Real,1}, k::Int)
function predictKNN(dataset::Batch, instances::AbstractArray{<:Real,2}, k::Int)
function streamLearning_KNN(datasetFolder::String, windowSize::Int, batchSize::Int, k::Int)
```