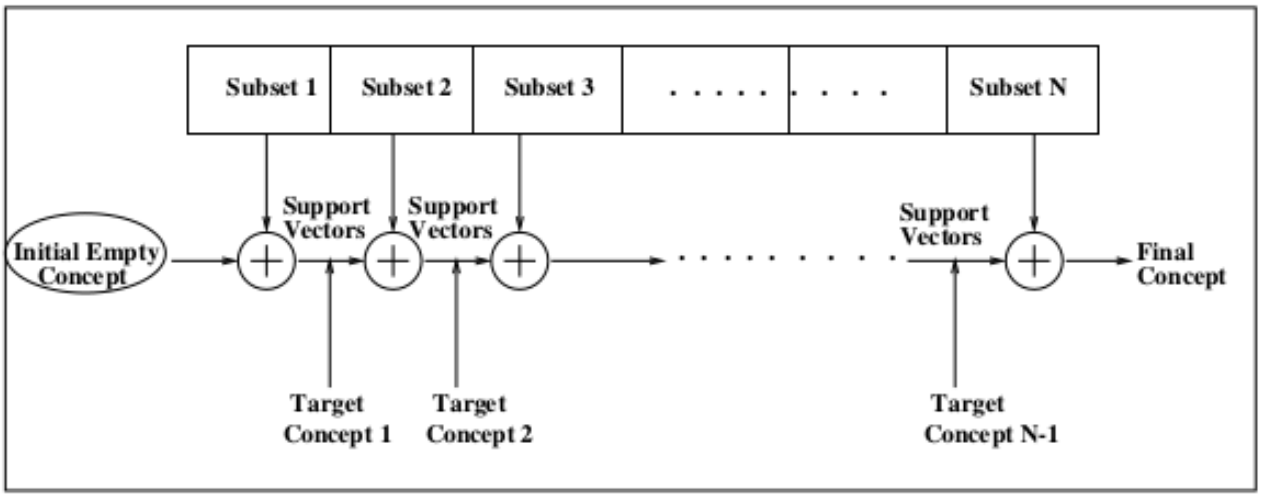
SVM incrementales

Modelos Avanzados de Aprendizaje Automático I

Los SVM incrementales son una extensión de los SVM tradicionales diseñados para actualizar el modelo de manera eficiente a medida que llegan nuevos datos o cuando estos están disponibles, sin necesidad de reentrenar el modelo desde cero, o sencillamente para reducir el tiempo de entrenamiento al entrenar con un conjunto de datos muy grande. Este enfoque es particularmente útil en aplicaciones donde los datos constituyen un conjunto de gran tamaño, no están disponibles todos, o bien estos son generados de forma continua.

Para utilizar un SVM en estos escenarios, la opción más común, como se explica en clase de teoría, es entrenar un SVM con cada lote de datos, a los que se unen los vectores de soporte resultado del anterior entrenamiento. De esta forma, los vectores de soporte constituyen un "resumen" de los datos previos, que se va uniendo a cada lote de datos nuevo. Este esquema se puede ver resumido en la siguiente figura:



En este ejercicio se llevará a cabo una implementación básica de este modelo. Para ello, se desarrollará el código necesario para trabajar con lotes de datos o *batches*, y se hará uso de la librería Scikit-Learn para entrenar los SVM.

Como se ha visto en ejercicios anteriores, así como en la asignatura FAA, los datos de entrenamiento se suelen proveer como un par (tupla de 2 elementos), donde el primer elemento es una matriz con las entradas, y el segundo elemento un vector o matriz con las salidas deseadas. A esto se conoce como *batch* o lote de datos. Para simplificar las funciones a realizar, en este ejercicio se crea un nuevo tipo, llamado *Batch*, y se define como:

```
Batch = Tuple{AbstractArray{<:Real,2}, AbstractArray{<:Any,1}}
```

Hay que tener en cuenta que en esta definición las salidas deseadas se están tomando como vectores, porque se trabajará con problemas de clasificación binarios. En otro tipo de problemas, o con más de dos clases, sería necesario cambiar esta definición de *batch*. Además, en estos ejercicios se supondrá que las instancias están situadas en las filas (atributos en columnas), por lo que el número de filas del primer elemento de la tupla debería ser igual al número de elementos del segundo.

Generalmente, cuando se hace una definición de un tipo, se suelen crear funciones que operen con ese tipo, para poder abstraer al programador de cómo están dispuestos los datos en ese tipo. Por ejemplo, se creará una función *batchInputs* que devolverá el primer elemento del *batch*, para que el programador no tenga que saber que, en realidad, el tipo *Batch* es una tupla donde las entradas son el primer elemento. Por este motivo, las primeras funciones a desarrollar en este ejercicio estarán orientadas a operar con el tipo *Batch*. A la hora de unir *batches*, en las funciones a desarrollar en este ejercicio y el siguiente se seguirá el criterio de que los datos más recientes se colocarán al final del *batch*, y, por lo tanto, los más antiguos serán los primeros. Es importante tener en cuenta que, para garantizar la repetibilidad de los resultados, en este ejercicio y en los siguientes, es importante unir conjuntos de datos respetando siempre este criterio. A pesar de que los SVM, desde un punto de vista teórico, son modelos determinísticos y no depende del orden de los datos, en la práctica los algoritmos de optimización que utilizan sí pueden depender del orden de los datos, y puede haber problemas de redondeo dependiendo de ese orden. Por ese motivo, como se indica, es importante seguir el criterio aquí descrito.

Además, en este ejercicio se hará uso de los paquetes MLJ, LIBSVM y MLJLIBSVMInterface. Concretamente, se utilizarán las funciones *SVMClassifier*, *mach*, *fit!* y *predict*, y se hará uso de los campos en el interior del modelo para hallar los vectores de soporte. Además, se provee de una implementación sencilla de la función *predict* para que se pueda usar de forma indistinta la de los paquetes ScikitLearn y MLJ.

En este ejercicio, se pide realizar las siguientes funciones:

- *batchInputs*. Función que recibe un lote de datos y devuelve la matriz de entradas. Recibe como parámetros:
    - *batch*, de tipo *Batch*, con el lote de datos.

  Esta función devolverá el primer elemento de la tupla del *batch*, por lo tanto, deberá ser una matriz.

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

- *batchTargets* Función que recibe un lote de datos y devuelve el vector de salidas deseadas. Recibe como parámetros:

    - *batch*, de tipo *Batch*, con el lote de datos.

  Esta función devolverá el segundo elemento de la tupla del *batch*, por lo tanto, deberá ser un vector.
  Para el desarrollo de esta función no se permite el uso de bucles.
- *batchLength*. Función que recibe un lote de datos y devuelve el número de instancias en este lote de datos. Recibe como parámetros:

    - *batch*, de tipo *Batch*, con el lote de datos.

  Esta función devolverá el número de instancias, que se puede leer como el número de filas de la matriz de entradas (leída con *batchInputs*), dado que las instancias están dispuestas en filas, o el número de elementos del vector de salidas deseadas (leído con *batchTargets*). En consecuencia, deberá devolver un valor entero positivo.
  Para el desarrollo de esta función no se permite el uso de bucles.
- *selectInstances*. Función que recibe un lote de datos y devuelve otro lote de datos, con las instancias indicadas. Recibe como parámetros:

    - *batch*, de tipo *Batch*, con el lote de datos.

    - *indices*, de tipo *Any*, con los índices a tomar del lote de datos.

  Esta función devolverá otro lote de datos, para lo cual tomará las entradas (con *batchInputs*) y salidas deseadas (con *batchTargets*) del lote de datos, selecciona las instancias (filas) correspondientes de las entradas y los elementos correspondientes de las salidas deseadas, y devuelve un nuevo lote de datos con esto, es decir, una nueva tupla. Es decir, deberá devolver un elemento de tipo *Batch*.
  Para el desarrollo de esta función no se permite el uso de bucles.
- *joinBatches*. Función que recibe dos lotes de datos y devuelve otro lote de datos, con la unión de los dos lotes, en el mismo orden en que se han pasado los lotes. Recibe como parámetros:

    - *batch1*, de tipo *Batch*, con el lote de datos a situar en la parte superior de la matriz de entradas e inicial del vector de salidas deseadas.

- *batch2*, de tipo *Batch*, con el lote de datos a situar en la parte interior de la matriz de entradas y final del vector de salidas deseadas.

Esta función devolverá un elemento de tipo *Batch*, con el resultado de concatenar las entradas y las salidas deseadas de ambos lotes de datos. Esto se puede hacer mediante la función *vcat*, siendo el primer elemento el que se pone al principio (el de *batch1*), y el segundo el que se pone al final (el de *batch2*).

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

- *divideBatches*. Función básica que, dado un conjunto de datos (que, en realidad, también es un *batch*), permite dividirlo en distintos lotes. Recibe como parámetros:

  - *dataset*, de tipo *Batch*, con el conjunto de datos.

  - *batchSize*, de tipo *Int*, con el tamaño del lote. De esta forma, todos los lotes tendrán este tamaño excepto el último, que tendrá habitualmente una cantidad inferior.

  Esta función también recibe el siguiente parámetro opcional:

  - *shuffleRows*, de tipo *Bool*, con un valor que indica si se quiere desordenar las instancias antes de dividirlas en lotes, o bien si esta división en lotes se hará siguiendo el orden en el que aparecen en las matrices de entradas y salidas deseadas (es decir, las primeras instancias estarán como primeras en el primer lote, y de la misma manera con el resto).

  Esta función debe devolver un vector donde cada elemento sea de tipo *Batch*, con todos los lotes. Funciones que pueden ser útiles para desarrollar esta función: *shuffle* (dentro del paquete *Random*), *partition* (dentro del paquete *Base.Iterators*), además de las funciones desarrolladas anteriormente para operar con *batches*, en concreto la función *selectInstances*.

  Para el desarrollo de esta función no se permite el uso de bucles. Para evitarlo, se puede hacer una operación de *Array comprehension* sobre el resultado de llamar a la función *partition*.

Una vez terminadas las funciones básicas para operar con lotes, se implementarán las funciones relativas al entrenamiento de un SVM incremental, que en realidad es una única función sobrecargada:

- *trainSVM*. Función que permite entrenar un SVM utilizando MLJ con un lote de datos. Tiene la particularidad de que devuelve, además del modelo entrenado, los vectores de soporte

resultado de este entrenamiento, y también que es capaz de recibir un lote adicional con los vectores de soporte de un entrenamiento previo. Recibe como parámetros:

- *dataset*, de tipo *Batch*, con el conjunto de entrenamiento.
- *kernel*, de tipo *String*, con el *kernel* a utilizar.
- *C*, de tipo *Real*, con el valor del hiperparámetro C.

Esta función recibe también como parámetros opcionales:

- *degree*, *gamma*, *coef0*, los tres de tipo *Real*, con los valores de estos hiperparámetros, que se usarán dependiendo del *kernel* especificado.
- *supportVectors*, de tipo *Batch*, con un lote de datos con los vectores de soporte de un entrenamiento anterior.

Esta función debe, en primer lugar, concatenar los vectores de soporte pasados como argumento con el conjunto de entrenamiento, con la función *joinBatches*. Según el criterio utilizado de tener los datos antiguos en las primeras filas, estas estarán formadas por los vectores de soporte, y el conjunto de entrenamiento estará al final. Además, no se deberá desordenar ninguna de estas matrices. De esta manera, se crea el lote de datos que se utilizará para entrenar el SVM.

Una vez se tienen los datos de entrenamiento, se crea el SVM con la función *SVMClassifier*. De esta forma, la creación del SVM será la siguiente:

```
model = SVMClassifier(
    kernel =
        kernel=="linear"  ? LIBSVM.Kernel.Linear :
        kernel=="rbf"     ? LIBSVM.Kernel.RadialBasis :
        kernel=="poly"    ? LIBSVM.Kernel.Polynomial :
        kernel=="sigmoid" ? LIBSVM.Kernel.Sigmoid : nothing,
    cost   = Float64(C),
    gamma  = Float64(gamma),
    degree = Int32(  degree),
    coef0  = Float64(coef0));
```

Una vez creado, es necesario crear el objeto de tipo machine, que actúa como enlace entre el modelo y el *dataset*, de la siguiente manera:

```
mach = machine(model, MLJ.table(batchInputs(dataset)), categorical(batchTargets(dataset)));
```

Es importante tener en cuenta que, en esta llamada a la función *machine*, el *dataset* pasado no deberá ser el original recibido como parámetro, sino que este habrá sido modificado añadiendo el conjunto de vectores de soporte. Para la llamada a esta función de entrenamiento, se pueden usar las funciones *batchInputs* y *batchTargets* para obtener los conjuntos de entradas y salidas deseadas. Posteriormente a la llamada a *machine*, este objeto se entrena con una llamada a la función *fit!* Como resultado, se tiene un modelo, que es un objeto que contiene varios campos. De todos estos campos, el que es más interesante para este ejercicio es el que contiene los índices en el conjunto de entrenamiento de los vectores de soporte como resultado de este entrenamiento, como un vector de valores enteros. Además, es interesante, de cara al siguiente ejercicio, tomar este vector de índices con los valores ordenados. De esta forma, la creación de este vector deberá ser la siguiente:

```
indicesNewSupportVectors = sort( mach.fitresult[1].SVs.indices );
```

Esta función debe devolver una tupla con tres valores. El primero será el modelo entrenado (el objeto *mach*), el segundo, un lote de datos con los vectores de soporte, y el tercero será una nueva tupla con dos vectores de índices que referencian qué vectores de soporte referencian al lote de datos original, y al conjunto de vectores de soporte pasado como parámetro. Es decir, esta función debe devolver una tupla con 3 valores, por este orden:

- Modelo entrenado (objeto de tipo *Machine*).
- Lote de datos (de tipo *Batch*) con los vectores de soporte de este entrenamiento (entradas y salidas deseadas). Es posible que algún vector de soporte no forme parte del conjunto de entrenamiento inicial pasado como parámetro, sino del conjunto de vectores de soporte pasado como parámetro.
- Tupla que será utilizada en un ejercicio posterior, con 2 vectores, por este orden (siguiendo el mismo criterio de tratar primero los datos más antiguos):
  - Vector de valores enteros (tipo *Integer*), con los índices de las instancias del conjunto de vectores de soporte pasados como parámetro que han sido vectores de soporte en este nuevo entrenamiento.
  - Vector de valores enteros (tipo *Integer*), con los índices de las instancias del conjunto de entrenamiento pasado como parámetro que han sido vectores de soporte en este nuevo entrenamiento.

Para hacer esta función, una vez entrenado el SVM, se tomarán los índices de los vectores de soporte del modelo entrenado de la forma especificada más arriba. De este vector de índices,

es necesario separar cuáles se corresponden con datos del conjunto original, y cuáles se corresponden con datos del conjunto de vectores de soporte pasados como argumento. Si N es el número de instancias del conjunto de vectores de soporte pasado como argumento:

- Los índices que se corresponden al conjunto de vectores de soporte pasado como parámetro son los valores de *indicesNewSupportVectors* con valor menor o igual que N.
- Los índices que se corresponden al conjunto de entrenamiento pasado como parámetro son los valores de *indicesNewSupportVectors* con valor mayor que N. A este valor hay que restarle N para que se corresponda con los vectores de soporte pasados como parámetro.

Una vez que se tienen estos dos vectores de índices, el conjunto de vectores de soporte a devolver como segundo elemento de la tupla se creará concatenando (función *joinBatches*) los datos correspondientes a los elementos que fueron seleccionados como vectores de soporte procedentes del conjunto de vectores de soporte pasados como parámetro, y los procedentes del conjunto de datos original, por ese orden. Para ello, se pueden usar las funciones *joinBatches* y *selectInstances*.

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

- *trainSVM*. Esta función permite entrenar un SVM de forma incremental, es decir, a partir de un vector de lotes de datos. Recibe como parámetros:

  - *batches*, de tipo *AbstractArray{<:Batch,1}*, el vector de lotes de datos para entrenar de forma incremental.
  - *kernel*, de tipo *String*, con el *kernel* a utilizar.
  - *C*, de tipo *Real*, con el valor del hiperparámetro C.

  Esta función recibe también como parámetros opcionales:

  - *degree*, *gamma*, *coef0*, los tres de tipo *Real*, con los valores de estos hiperparámetros, que se usarán dependiendo del *kernel* especificado.

  Esta función comenzará con la definición de un *batch* de vectores de soporte vacío, tras lo cual se creará un bucle que iterará por todos los lotes de datos, siguiendo su orden en el vector. En cada iteración del bucle únicamente se llamará a la función anterior (*trainSVM*) pasando como parámetros el lote correspondiente a esa iteración, la configuración del SVM, y el *batch* de vectores de soporte que se tenga. Como salida de esa función se tomará el

modelo entrenado, y se sobreescribirá el *batch* de vectores de soporte.

Finalmente, la salida de esta función será el último modelo entrenado (de tipo *Machine*), es decir, en la última iteración del bucle anterior. Es importante tener en cuenta que, si este modelo ha sido asignado a una variable que se crea en cada iteración de un bucle, esta no estará disponible fuera del bucle, por lo que no se podrá devolver fuera del bucle. Para solucionar esto, lo más sencillo es crear la variable con ese mismo nombre antes de definir el bucle, y, de esta manera, en cada iteración se sobreescribirá su valor, estando disponible una vez haya terminado el bucle.

Para el desarrollo de esta función se permite el uso de un único bucle.

En este ejercicio, es interesante experimentar con distintos *datasets*, con un número de instancias que no sea muy bajo, como "agaricus-lepiota", "twonorm", "ring" o "hypothyroid". Para cada uno, se puede comparar los resultados obtenidos al entrenar un SVM de la forma habitual, o un SVM incremental. Para comparar correctamente ambos casos, se dividirá el conjunto de datos en *batches*, reservando el último para hacer test. En el primer caso, se entrena el SVM con todos los lotes unidos a excepción del último. En el segundo caso, se entrena el ISVM de forma incremental excluyendo el último lote de datos. En ambos casos, el modelo devuelto se prueba con el último lote de datos. Para el caso de ISVM, probar con distintos tamaños de lotes.

- ¿Qué tamaño de lote devuelve una precisión en test más cercana a la del entrenamiento clásico de SVM?

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 batchInputs(batch::Batch)
function batchTargets(batch::Batch)
function batchLength(batch::Batch)
function selectInstances(batch::Batch, indices::Any)
function joinBatches(batch1::Batch, batch2::Batch)
function divideBatches(dataset::Batch, batchSize::Int; shuffleRows::Bool=false)
function trainSVM(dataset::Batch, kernel::String, C::Real;
        degree::Real=1, gamma::Real=2, coef0::Real=0.,
        supportVectors::Batch=( Array{eltype(dataset[1]),2}(undef,0,size(dataset[1],2)) ,
                                Array{eltype(dataset[2]),1}(undef,0) ) )
function trainSVM(batches::AbstractArray{<:Batch,1}, kernel::String, C::Real;
        degree::Real=1, gamma::Real=2, coef0::Real=0.)
```