# Librerías de Aprendizaje Automático en Julia

## Aprendizaje Automático

Julia ofrece un rico ecosistema de distintas librerías que permiten ejecutar una gran cantidad de algoritmos de Aprendizaje Automático. En este aspecto, una de las más conocidas es la librería Scikit-learn. Esta es una librería para aprendizaje automático de software libre desarrollada para el lenguaje de programación Python, cuya primera versión es del año 2010. Implementa una gran cantidad de modelos de aprendizaje automático, referidos a tareas como clasificación, regresión, *clustering* o reducción de la dimensionalidad. Estos modelos incluyen Máquinas de Vectores de Soporte (SVM), árboles de decisión, *random forests*, o k-means. Actualmente es una de las librerías más utilizadas en el campo del aprendizaje automático, por la gran cantidad de funcionalidades que ofrece así como por la facilidad de su uso, puesto que provee de una interfaz uniforme para el entrenamiento y uso de modelos. La documentación de esta librería está disponible en https://scikit-learn.org/stable/

Para Julia, la librería ScikitLearn.jl implementa esta interfaz y los algoritmos que contiene la librería scikit-learn, dando soporte tanto a los modelos propios de Julia como a los de la librería scikit-learn. Esto último lo realiza por medio de la librería PyCall.jl, que permite ejecutar código escrito en Python. En https://scikitlearnjl.readthedocs.io/en/latest/ puede encontrarse documentación de esta librería. Sin embargo, para evitar posiblers problemas de incompatibilidades entre Julia, PyCall y ScikitLearn, para hacer este ejercicio se utilizará otra librería distinta.

La librería a utilizar es MLJ (*Machine Learning in Julia*), que no es propiamente una librería, sino un *framework* que permite utilizar distintas librerías relacionadas mediante una interfaz común. De esta manera, los nombres de las funciones para crear y entrenar modelos van a ser las mismas independientemente de los modelos que se quieran desarrollar. En las prácticas de esta asignatura, además de RR.NN.AA. y DoME, se utilizarán los siguientes modelos, disponibles en la librería:

- Máquinas de Soporte Vectorial
- kNN
- Árboles de decisión

Para poder utilizar estos modelos, en primer lugar es necesario importar la librería (*using MLJ*, para lo cual debe estar previamente instalada con *import Pkg; Pkg.add("MLJ")*). De la misma manera, será necesario tener instalados los paquetes que contienen los algoritmos a ejecutar (*LIBSVM*,

*NearestNeighborModels*, *DecisionTreeClassifier*), así como los paquetes que contienen las interfaces entre estos algoritmos y el *framework* MLJ (*MLJLIBSVMInterface*, *MLJDecisionTreeInterface*). Para importar los modelos que se van a emplear, se puede usar la macro *MLJ.@load*. De esta forma, las siguientes líneas importan respectivamente los 3 primeros modelos antes mencionados que se van a utilizar en las prácticas de esta asignatura:

```
SVMClassifier = MLJ.@load SVC pkg=LIBSVM verbosity=0
kNNClassifier = MLJ.@load KNNClassifier pkg=NearestNeighborModels verbosity=0
DTClassifier  = MLJ.@load DecisionTreeClassifier pkg=DecisionTree verbosity=0
```

Como se puede ver, cada modelo se carga de un paquete distinto. El comando *verbosity=0* del final se utiliza solamente para indicar que no se escriba por pantalla el resultado de esta importación. De esta forma, se han definido 3 funciones, para crear cada uno de los 3 modelos. Cada función recibe como parámetros los parámetros propios del modelo. A continuación se muestran 3 ejemplos, uno para cada tipo de modelo que se va a usar en estas prácticas de esta asignatura:

```
model = SVMClassifier(kernel= LIBSVM.Kernel.RadialBasis,
      cost = 1., gamma = 2., degree = Int32(3));
model = DTClassifier(max_depth = 4, rng=Random.MersenneTwister(1))
model = kNNClassifier(K = 3) ;
```

Al crear un kNN, el hiperparámetro principal es K, que define el número de vecinos. Para árboles de decisión, el hiperparámetro principal es *max_depth*, que indica la profundidad máxima del árbol. En el caso de los árboles de decisión, como se puede ver, hay otro parámetro denominado *rng*. Este parámetro controla la aleatoriedad en una parte concreta del proceso de construcción del modelo. En el caso concreto de los árboles de decisión, esta aleatoriedad está en la selección de características para dividir un nodo del árbol. La librería *DecisionTree* utiliza en esta parte el generador de números aleatorios, que se actualiza con cada llamada, con lo que distintas llamadas a esta función (junto a sus posteriores llamadas a la función *fit!*) para entrenar el modelo darán lugar a modelos distintos. Para controlar la aleatoriedad de este proceso y que este sea determinístico, lo mejor es darle un valor entero como se puede ver en el ejemplo. De esta forma, la creación de un modelo con un conjunto de entradas y salidas deseadas y un conjunto de hiperparámetros dado es un proceso determinístico. En general, es más recomendable poder controlar la aleatoriedad de todo el proceso de desarrollo de modelos (validación cruzada, etc.) mediante una semilla aleatoria que se fije al inicio de todo el proceso. Sin embargo, en estas prácticas se utilizará el *keyword rng* para este modelo.

Las SVM tienen un conjunto de hiperparámetros más complejo, que depende de la función de *kernel* a utilizar. En primer lugar, el hiperparámetro C controla el equilibrio entre margen y error de clasificación. Valores bajos permiten más errores, valores altos ajustan más el modelo. Para usar este hiperparámetro, se tiene el keyword *cost* en la llamada a la función *SVMClassifier*. Además de este hiperparámetro, es necesario fijar el kernel que se va a utilizar. Para indicar el kernel a utilizar en la llamada a la función *SVMClassifier*, se tiene el *keyword kernel*, que puede tomar uno de los siguientes valores, dados por la librería LIBSVM:

- LIBSVM.Kernel.Linear
- LIBSVM.Kernel.RadialBasis
- LIBSVM.Kernel.Sigmoid
- LIBSVM.Kernel.Polynomial

Adicionalmente, según el *kernel* a utilizar, se tendrán unos hiperparámetros u otros:

- Kernel lineal: No tiene más hiperparámetros además de C.
- Kernel RBF (*Radial Basis Function*). Además de C, tiene como hiperparámetro *gamma*, que controla la influencia de los puntos de soporte.
- Kernel sigmoidal. Además de C, *gamma* ajusta la influencia de los puntos y *coef0* controla el término independiente. Con este kernel, el SVM funciona similar a una red neuronal, donde *gamma* y *coef0* afectan la forma de la función de decisión.
- Kernel polinómico. Además de C, *degree* define el grado del polinomio, *gamma* ajusta la influencia de los puntos y *coef0* controla el término independiente.

Valores típicos son 0.001, 0.1, 1, 10, 100 y 1000.

La siguiente tabla muestra los distintos hiperparámetros, junto con el kernel que los usa, así como los valores típicos que suelen tomar. Recordad que en la llamada a la función *SVMClassifier* estos hiperparámetros toman el mismo nombre, excepto C, que toma el nombre *cost*, como se muestra en el ejemplo anterior. Además, es importante que el tipo del argumento pasado sea exactamente el que se especifica en la tabla, o de lo contrario se tendrá un error. Para ello, lo mejor es forzar el tipado a la hora de pasar el hiperparámetro como argumento.

| Hiperparámetro | kernels | Valores típicos | Tipo en LIBSVM |
|---|---|---|---|
| C | Lineal, RBF, sigmoidal, polinómico | 0.001, 0.1, 1, 10, 100, 1000 | Float64 |
| gamma | RBF, sigmoidal, polinómico | 0.1, 0.01, 0.001, 0.0001 | Float64 |
| coef0 | sigmoidal, polinómico | 0, 1, 5, 10 | Int32 |
| degree | polinómico | 2, 3, 4, 5 | Float64 |

A pesar de que el modelo básico de SVM solamente permite la clasificación en dos clases, la implementación de SVM en la librería MLJ ya permite la clasificación multiclase, por lo que no es necesario utilizar una estrategia "uno contra todos" para estos casos.

Una vez creados los modelos, es necesario crear un objeto *machine*. Este objeto representa un contenedor que asocia un modelo con datos y administra su entrenamiento y predicción. Es un concepto clave en MLJ que facilita el ajuste (mediante la función *fit!*) y la predicción (mediante la función *predict*) de modelos. Este objeto, por lo tanto, tiene tres componentes principales:

- Modelo: Especifica qué algoritmo usar, creado anteriormente, sin datos ni estado.
- Datos: Define las características y las etiquetas si el modelo es supervisado.
- Estado interno: Guarda los parámetros aprendidos después del entrenamiento.

Para crear este objeto, se puede utilizar la función *machine*, indicándole como parámetros el modelo creado, el conjunto de entradas y el conjunto de salidas deseadas. Con relación a estos dos últimos parámetros, es importante tener en cuenta que esta función no acepta *Arrays* como entradas, sino que deben ser convertidas a algún tipo de tablas de las soportadas por MLJ, como *Tables.table*, *DataFrame* o *NamedTuple*. Si los datos se tienen como *Arrays*, como se está haciendo en estas prácticas, la creación del objeto machine se puede hacer con la siguiente línea:

```
mach = machine(model, MLJ.table(trainingInputs), categorical(trainingTargets));
```

Como se puede ver, la matriz de entradas se convierte a una tabla, y el vector de salidas deseadas a una lista de valores categóricos, puesto que en este ejercicio se resolverán problemas de clasificación. Es importante darse cuenta de que esta variable *targets* con las salidas deseadas es un vector y no una matriz. Cada elemento del vector se corresponderá con la etiqueta del patrón correspondiente, y puede tener cualquier tipo: entero, *string*, etc. A pesar de que algunos modelos aceptan las salidas deseadas con la codificación vista *one-hot-encoding*, otros no la aceptan, por lo que en esta práctica esta función utilizará como salidas deseadas un vector donde cada elemento es la etiqueta, al contrario que en el caso de las RR.NN.AA. Además, para evitar posibles problemas en

algunas librerías, los elementos de este vector se convertirán a *String* antes de esta llamada.

Una vez creado el objeto *machine*, este se puede entrenar mediante la función *fit!*, de la siguiente manera:

```
MLJ.fit!(mach, verbosity=0)
```

Como se puede ver, esta función solamente recibe de forma obligatoria el objeto *machine*, sin necesidad de pasar los datos de entrenamiento, puesto que ya están dentro del propio objeto machine. Adicionalmente, el parámetro opcional *verbosity=0* indica que no se ponga por pantalla el proceso de entrenamiento.

- ¿Qué indica el hecho de que el nombre de esta función termine en *bang* (!)?

Al contrario de lo que ocurría con la librería Flux, en la que era necesario escribir el bucle de entrenamiento de la RNA, en esta librería el bucle ya está implementado, y se llama automáticamente al ejecutar la función *fit!*. Por lo tanto, no es necesario escribir el código con el bucle de entrenamiento.

Una cuestión importante a tener en cuenta es la disposición de los datos que se van a utilizar. Como se ha mostrado en prácticas anteriores, para entrenar una RNA los patrones deberán estar dispuestos en columnas, y en la matriz de entradas cada fila será un atributo. Fuera del mundo de las RR.NN.AA., y por lo tanto con el resto de técnicas que se van a utilizar en esta asignatura, los patrones se suele suponer que están dispuestos en filas, y por lo tanto cada columna en la matriz de entradas se corresponde con un atributo, siendo una forma mucho más intuitiva.

- ¿Qué condición deben cumplir la matriz de entradas y el vector de salidas deseadas que se le pasen como argumento a esta función?

Finalmente, una vez el modelo haya sido entrenado, este puede ser utilizado para realizar predicciones. Esto se realiza mediante la función *predict*. A continuación se muestra un ejemplo de uso:

```
testOutputs = MLJ.predict(mach, MLJ.table(testInputs));
```

Como se puede ver, es necesario pasar como parámetro el objeto *machine* y la matriz de entradas, convertida. Como resultado, en problemas de clasificación devuelve una variable de tipo distinto según el modelo y la librería que se esté usando. Para los modelos de las librerías utilizadas en este ejercicio:

- Para SVM, *predict* devuelve una variable de tipo *CategoricalArray*, que se puede comparar directamente con el vector de salidas deseadas. Por lo tanto, no es necesario hacerle ningún tipo de postprocesado.

- Para Árboles de Decisión y kNN, *predict* devuelve una variable *UnivariateFiniteArray*, que representa una distribución de probabilidad sobre las clases. Para convertir esta variable en una categórica y poder compararla con las salidas deseadas, se puede usar la función *mode*, para obtener la clase más probable.

El modelo que se está usando es una estructura en memoria con distintos campos, y puede ser de gran utilidad consultar los contenidos de esos campos. El objeto *machine* es una estructura que contiene el modelo, los datos, y los resultados del entrenamiento. Por tanto, al modelo se puede acceder a través del objeto *machine* o, más fácilmente, a través de la variable *model*. Por ejemplo, si se quiere entrenar un SVM se puede acceder a un hiperparámetro de forma indistinta con:

```
model.gamma
```

```
mach.model.gamma
```

Por su parte, para consultar los parámetros resultado del entrenamiento, se puede hacer de varias formas. Uno de los parámetros más interesantes, en el caso de los SVM, es poder consultar qué instancias han sido utilizadas como vectores de soporte. Esto se puede hacer de cualquiera de estas dos maneras:

```
mach.fitresult[1].SVs.indices
```

```
fitted_params(mach)[:libsvm_model].SVs.indices
```

En este ejercicio se desarrollará una única función que permite entrenar los 3 modelos diferentes a través de la librería MLJ, y, adicionalmente, redes de neuronas artificiales y DoME utilizando las funciones desarrollada en los ejercicios anteriores. Este entrenamiento se realizará mediante una validación cruzada. En cada *fold*, se entrenará el modelo especificado y se calcularán las métricas en el conjunto de test. Además, al igual que en el ejercicio anterior, es de utilidad crear una matriz de confusión con la distribución de las instancias en los conjuntos de test. En este caso es más sencillo que el anterior, puesto que, al ser los métodos determinísticos, sólo se creará una matriz de confusión en cada *fold*, y la matriz de confusión final será la suma de todas ellas. De todas maneras, las consideraciones dadas en el ejercicio anterior siguen siendo aplicables aquí, en especial la de que las métricas que se pudieran extraer de esta matriz de confusión global en general no van a coincidir

con las calculadas por medio de la validación cruzada.

La descripción de la función a desarrollar es la siguiente:

- Desarrollar una función llamada *modelCrossValidation* para que, además de entrenar redes de neuronas, también se realice validación cruzada sobre SVM, árboles de decisión, kNN y DoME. Los argumentos que debe recibir esta función son los siguientes:

    - El primero, *modelType*, de tipo *Symbol*, contendrá un indicador del modelo a entrenar. Este símbolo será igual a *:ANN* para el caso de entrenar redes de neuronas, :DoME para el algoritmo DoME, :SVC para SVM, *:DecisionTreeClassifier* para árboles de decisión y *:KNeighborsClassifier* para kNN.

    - El segundo, *modelHyperparameters*, contiene un objeto de tipo *Dict* con los hiperparámetros del modelo. Las entradas de los hiperparámetros en esta variable tipo *Dict* deberán ser de tipo *String* o de tipo *Symbol*.

      Para el caso de los hiperparámetros de redes de neuronas, descritos en el ejercicio anterior, estos deberán tener el mismo nombre. Además, tened en cuenta que la mayoría son opcionales, siendo *topology* el único obligatorio. Por lo tanto, es posible que falte algún hiperparámetro. Para saber si una variable de tipo *Dict* contiene una clave o no, se puede usar la función *haskey*. Otra función que puede ser útil para tomar el valor de una clave que podría no estar presente en un objeto de tipo Dict es *get*.

      Para el resto de los modelos, a continuación se listan los hiperparámetros que habrá que definir, con sus nombres exactos:

        - DoME (*:DoME*): *maximumNodes*, que contiene el número máximo de nodos que tendrá el árbol, y toma valores enteros positivos.

        - SVM (*:SVC*): *C*, *kernel*, *degree*, *gamma* y *coef0*. El hiperparámetro *kernel* podrá tomar los valores (de tipo *String*) "linear", "rbf", "sigmoid" y "poly". Los hiperparámetros *degree*, *gamma* y *coef0* describen los valores de los parámetros propios de cada kernel, y, según el que se utilice, alguno puede ser ignorado. Por ejemplo, el kernel “poly” utiliza el grado del polinomio (*degree*), la pendiente (*gamma*), y el término independiente (*coef0*), mientras que el kernel “sigmoid” utiliza *gamma* y *coef0*. El hiperparámetro C se usa en

todos los casos. Tened en cuenta que el hiperparámetro C se pasa con el keyword *cost*, el kernel mediante los valores definidos en la librería LIBSVM (*LIBSVM.Kernel.Linear*, *LIBSVM.Kernel.RadialBasis*, *LIBSVM.Kernel.Sigmoid*, y *LIBSVM.Kernel.Polynomial*), y que, para evitar errores, es conveniente forzar el tipo de los parámetros. Por ejemplo, para crear un SVM polinómico se puede hacer mediante:

```
model = SVMClassifier(
      kernel = LIBSVM.Kernel.Polynomial,
      cost = Float64(C),
      gamma = Float64(gamma),
      degree = Int32(degree),
      coef0 = Float64(coef0));
```

- Árboles de decisión (*:DecisionTreeClassifier*): *max_depth*, que contiene la profundidad máxima de los árboles. Además, al crear el modelo se debe especificar la semilla aleatoria mediante *rng*. Esta semilla debe tomar un valor de 1 para garantizar la repetibilidad de los resultados.

- kNN (*:KNeighborsClassifier*): *n_neighbors*, con el valor de k, que especifica el número de vecinos a tomar.

- *dataset*, de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{<:Any,1}}*, con una tupla con dos valores. El primero de ellos es la matriz de entradas. Al contrario que en el entrenamiento de Redes de Neuronas, el uso de *Float32* no está tan extendido en la librería, siendo *Float32* o *Float64* igualmente utilizados, dependiendo de la precisión buscada. Por este motivo, no es necesario convertir las entradas a *Float32*. El segundo valor de esta tupla será el vector de salidas deseadas.

- *crossValidationIndices*, de tipo *Array{Int64,1}*. Es importante tener en cuenta que, al igual que en la práctica anterior, la división de los patrones en cada *fold* es necesario hacerla fuera de esta función, porque de esta manera se permite que esta misma división se utilice al entrenar otros modelos. De esta forma, se realizará validación cruzada con los mismos datos y las mismas particiones en todos los casos.

Esta función comenzará comprobando si se desea entrenar redes de neuronas (examinando el parámetro *modelType*). Si es así, se hace una llamada a la función *ANNCrossValidation* con

los parámetros indicados en el parámetro *modelHyperparameters*, teniendo en cuenta que muchos de ellos podrían no estar definidos en esa variable. Como se ha puesto anteriormente, la función *haskey* permite comprobar si un objeto de tipo *Dict* contiene una clave o no, lo que, en este caso, se corresponde con que un hiperparámetro haya sido definido o no. Otra función que se puede utilizar en lugar de *haskey* es *get*. Una vez llamada la función *ANNCrossValidation*, se devuelve el resultado de esta llamada, con lo que, en caso de entrenar redes de neuronas, se saldría de la función en este punto.

En caso de querer entrenar uno de los otros modelos, se procede de forma similar a la función del ejercicio anterior: creando 7 vectores, para los resultados para cada una de las 7 métricas en cada *fold*, además de un *Array* bidimensional para almacenar la matriz de confusión, con valores inicialmente a 0.

Una modificación importante es, en los modelos de la librería a usar, antes de entrenar ningún modelo, transformar el vector de salidas deseadas en un vector de cadenas de texto, para evitar cualquier posible error con las distintas librerías propias de los modelos. Esto se puede hacer, sencillamente, con la siguiente línea:

```
targets = string.(targets);
```

Además, será necesario calcular el vector de clases, al igual que en el ejercicio anterior, con

```
classes = unique(targets);
```

Una vez hechas estas sencillas operaciones, se puede comenzar con el bucle de la validación cruzada. En cada iteración, en primer lugar se calculan las matrices de entradas de entrenamiento y test, y los vectores de salidas deseadas de entrenamiento y test (de tipo *AbstractArray{<:Any,1}*), para, posteriormente, crear el modelo con los hiperparámetros especificados y entrenarlo. Esto se realizará de forma distinta para el algoritmo DoME y los modelos pertenecientes a la librería Scikit-Learn. Para el algoritmo DoME, únicamente será necesario llamar a la función *trainClassDoME* desarrollada en el ejercicio 4, y ya devolverá las etiquetas del conjunto de test. Para los modelos de la librería MLJ, será necesario crear el modelo con la función correspondiente (*SVMClassifier*, *DTClassifier* o *kNNClassifier*) indicándole los hiperparámetros, después llamar a *machine* y a *fit!* para entrenar el modelo, y, finalmente, llamar a *predict* y *mode* (esto último sólo para el caso de árboles de decisión y kNN) para calcular las etiquetas del conjunto de test. Este código a desarrollar deberá ser el

mismo para cada uno de los 3 tipos de modelos (SVM, Árboles de Decisión, kNN), con las salvedades de que la línea en la que se cree el modelo debe ser distinta para cada uno de los modelos, y la aplicación de la función *mode* de forma dependiente del modelo.

Una vez se tengan las etiquetas del conjunto de test, se calcularán las métricas y la matriz de confusión con la función *confusionMatrix*. Las métricas devueltas se asignarán a las posiciones correspondientes de los vectores, y la matriz de confusión obtenida se sumará a la matriz de confusión global en test. Como diferencia principal con el entrenamiento de RR.NN.AA. del ejercicio anterior, estos modelos deberán ser entrenados una única vez, al ser deterministas.

- Si se entrenan varias veces, ¿qué propiedades estadísticas tendrán los resultados de estos entrenamientos?

Como se ha descrito previamente, en el caso de usar técnicas como SVM, árboles de decisión, kNN o DoME, no se hará uso de la configuración *one-hot-encoding*. En estos casos, para calcular las métricas se hará uso de la función *confusionMatrix* desarrollada en una práctica anterior que acepta como entradas tres vectores (salidas, salidas deseadas y vector de clases) de tipo *AbstractArray{<:Any,1}*. Es **importante** usar la versión de la función que recibe el vector de clases.

- ¿Qué podría ocurrir si se utiliza la versión de la función que no recibe el vector de clases?

Esta función deberá devolver lo mismo que la realizada en el ejercicio anterior: una tupla con 8 valores. Los 7 primeros serán uno para cada métrica, y cada uno, a su vez, será una tupla con la media y la desviación típica de los valores de esa métrica en cada *fold*. Al igual que en el ejercicio anterior, estos 8 valores deberán ser, por este orden: precisión, tasa de error, sensibilidad, especificidad, VPP, VPN y F1. El octavo valor será la matriz de confusión global en test.

Una vez desarrollada esta función, se puede utilizar para evaluar distintas configuraciones, comparando los resultados de test obtenidos con cada una en la(s) métrica(s) que sean más apropiadas para el problema en cuestión. Por lo tanto, esta técnica no devuelve un modelo para poner en producción, sino una configuración. Es decir, la técnica de validación cruzada no genera un modelo final, sino que permite comparar distintos algoritmos y configuraciones para escoger

el modelo o configuración de hiperparámetros que devuelve los mejores resultados. Una vez escogido, para generar el modelo final es necesario entrenarlo desde 0 utilizando esta vez todos los patrones sin realizar validación cruzada, es decir, entrenar una única vez sin separar patrones para realizar test. De esta forma, se espera que el rendimiento de este modelo y configuración sea un poco superior al obtenido mediante validación cruzada puesto que se han utilizado más patrones para entrenarlo. Este es el modelo final que se utilizaría en producción, y del cual se puede obtener una matriz de confusión.

---

**Aprende Julia:**

Un tipo de Julia que es necesario para esta práctica es el tipo *Symbol*. Un objeto de este tipo puede ser cualquier símbolo que se quiera, simplemente escribiendo el nombre después de dos puntos (":"). En la función *modelCrossValidation* a desarrollar en esta práctica, se puede utilizar para indicar qué modelo se quiere entrenar, por ejemplo *:KNeighborsClassifier*, *:SVC*, *:DecisionTreeClassifier*, *:DoME* o *:ANN*.

Además, en esta práctica, a la función *modelCrossValidation* a definir es necesario pasar parámetros que son dependientes del modelo. Para hacer esto, lo más sencillo es crear una variable de tipo *Dictionary* (en realidad el tipo es *Dict*) que funciona de forma similar a Python. Por ejemplo, para especificar los parámetros de una RNA, se podría crear una variable de la siguiente manera:

```
modelHyperparameters = Dict("topology" => [5,3], "learningRate" => 0.01,
"validationRatio" => 0.2, "numExecutions" => 50, "maxEpochs" => 1000,
"maxEpochsVal" => 6);
```

Otra forma de definir esa variable podría ser la siguiente:

```
modelHyperparameters = Dict();

modelHyperparameters["topology"] = topology;

modelHyperparameters["learningRate"] = learningRate;

modelHyperparameters["validationRatio"] = validationRatio;

modelHyperparameters["numExecutions"] = numRepetitionsANNTraining;

modelHyperparameters["maxEpochs"] = numMaxEpochs;

modelHyperparameters["maxEpochsVal"] = maxEpochsVal;
```

Para leer los valores, se pueden utilizar los corchetes indicando el nombre del valor a leer, como por

ejemplo:

```
modelHyperparameters["topology"]
```

De la misma manera, se podría hacer algo similar para SVM, árboles de decisión, DoME y kNN. Un ejemplo, para los hiperparámetros de un SVM, podría ser el siguiente:

```
modelHyperparameters = Dict("C" => 1, "kernel" => "rbf", "gamma" => 2);
```

Otra forma de definir esa variable podría ser la siguiente:

```
modelHyperparameters = Dict();
modelHyperparameters["C"] = 1;
modelHyperparameters["kernel"] = "rbf";
modelHyperparameters["gamma"] = 2;
```

Otros kernel tendrían unos hiperparámetros distintos, como los ya mencionados *degree* y *coef0*.

Una vez dentro de la función a desarrollar, en la línea en la que se cree un SVM, esto se podría hacer de forma similar a:

```
if modelHyperparameters["kernel"] == "rbf"
      model = SVMClassifier(kernel = LIBSVM.Kernel.RadialBasis,
            cost = Float64(modelHyperparameters["C"]),
            gamma = Float64(modelHyperparameters["gamma"]));
```

De la misma manera, se podría hacer algo similar para árboles de decisión, kNN y DoME.

En estos ejemplos, se han especificado variables de tipo *Dict* donde las entradas son de tipo *String*. Para el desarrollo de este ejercicio, son igualmente válidas entradas de tipo *Symbol*, con los mismos nombres, por ejemplo:

```
modelHyperparameters = Dict(:C => 1, :kernel => "rbf", :gamma => 2);
```

---

**Firmas de las funciones:**

A continuación se muestra la firma de las función a realizar en este ejercicio.

```
function modelCrossValidation(modelType::Symbol, modelHyperparameters::Dict,
      dataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{<:Any,1}},
      crossValidationIndices::Array{Int64,1})
```