La librería ScikitLearn.jl

Aprendizaje Automático

La librería Scikit-learn 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 soporte Vectorial (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, pero cuyo uso es transparente para el usuario, que sólo necesita tener instalado ScikitLearn.jl para hacer uso de toda la funcionalidad de scikit-learn en Julia. En https://scikitlearnjl.readthedocs.io/en/latest/ puede encontrarse documentación de esta librería.

Como ya se ha dicho, esta librería ofrece una interfaz uniforme para entrenar distintos modelos. Esto se plasma en que 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., se utilizarán los siguientes modelos, disponibles en la librería scikit-learn:

- Máquinas de Soporte Vectorial
- Árboles de decisión
- kNN

Para poder utilizar estos modelos, en primer lugar es necesario importar la librería (*using ScikitLearn*, para lo cual debe estar previamente instalada con *import Pkg; Pkg.add("ScikitLearn")*) y los modelos. La librería scikit-learn ofrece más de 100 tipos

distintos de modelos. Para importar aquellos que se van a emplearse, se puede usar *@sk_import*. 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:

```
@sk_import svm: SVC

@sk_import tree: DecisionTreeClassifier

@sk_import neighbors: KNeighborsClassifier
```

A la hora de entrenar un modelo, el primer paso será generarlo. Esto se realiza con una función distinta para cada modelo. Esta 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 = SVC(kernel=”rbf”, degree=3, gamma=2, C=1);

model = DecisionTreeClassifier(max_depth=4, random_state=1)

model = KNeighborsClassifier(3);
```

En la documentación de la librería se puede encontrar una explicación acerca de los parámetros que aceptan cada una de estas funciones. En el caso particular de los árboles de decisión, como se puede ver, uno de estos parámetros se denomina *random_state*. Este parámetro controla la aleatoriedad en una parte concreta del proceso de construcción del árbol, concretamente en la selección de características para dividir un nodo del árbol. La librería Scikit-Learn 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 árbol de decisión 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.

Una vez creados, ya se pueden ajustar con la función *fit!* cualquiera de estos modelos.

- ¿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.

- Al igual que en RR.NN.AA., si se quieren entrenar varios modelos, será necesario desarrollar un bucle. ¿En qué parte del código (dentro o fuera del bucle) habrá que crear el modelo? ¿En qué modelos va a ser necesario entrenar varias veces y en cuáles una única vez? ¿Por qué?

Un ejemplo de uso de esta función se puede ver en la siguiente línea:

```
fit!(model, trainingInputs, trainingTargets);
```

Como se puede ver, el primer argumento de esta función es el modelo, el segundo es una matriz de entradas, y el tercero un vector de salidas deseadas. Es importante darse cuenta de que este parámetro 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.

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 = predict(model, testInputs);
```

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. Para ver qué campos tiene cada modelo, se puede escribir lo siguiente:

```
println(keys(model));
```

Según el tipo de modelo, habrá unos campos u otros. Por ejemplo, para un kNN se podrían consultar entre otros lo siguientes campos:

```
model.n_neighbors
```

```
model.metric
```

```
model.weights
```

Por su parte, para un SVM algunos campos interesantes podrían ser los siguientes:

```
model.C
```

```
model.support_vectors_
```

```
model.support_
```

En el caso de un SVM, una función especialmente interesante es *decisión_function*, que devuelve las distancias al hiperplano de los patrones pasados. Esto es útil, por ejemplo, para implementar una estrategia “uno contra todos” para realizar clasificación multiclase. A continuación se muestra un ejemplo de uso de esta función:

```
distances = decision_function(model, inputs);
```

- Para el caso de usar árboles de decisión o kNN no es necesaria una función correspondiente para realizar la estrategia “uno contra todos”, ¿por qué?

Sin embargo, la implementación de SVM en la librería Scikit-Learn ya permite la clasificación multiclase, por lo que no es necesario utilizar una estrategia “uno contra todos” para estos casos.

Por último, es necesario tener en cuenta que estos modelos suelen recibir entradas y salidas preprocesadas, siendo el preprocesado más común la normalización ya descrita en una

práctica anterior. Por lo tanto, las funciones de normalización desarrolladas deberán ser utilizadas también en los datos a usar por estos modelos.

En esta práctica, se pide:

- Haciendo uso del código desarrollado en las prácticas anteriores, desarrollar una función llamada *modelCrossValidation* que permita validar modelos en el problema de clasificación seleccionado usando las tres técnicas aquí descritas.
    - Esta función debe realizar una validación cruzada y utilizar las métricas que se consideren más apropiadas para el problema en cuestión. Esta validación cruzada puede hacerse modificando el código desarrollado en la práctica anterior.
- Esta función debe recibir los siguientes parámetros:
    - Algoritmo a entrenar, entre los 4 que se utilizan en esta asignatura, junto con sus parámetros. Los parámetros más importantes a especificar para cada técnica son:
        - RR.NN.AA: arquitectura (número de capas ocultas y número de neuronas en cada capa oculta) y función de transferencia en cada capa. En redes “poco profundas” como las utilizadas en esta asignatura, la función de transferencia tiene un impacto menor, por lo que se puede usar una estándar, como *tansig* o *logsig*. Otros parámetros a utilizar son la tasa de aprendizaje, la tasa de patrones usados para validación, el número máximo de ciclos de entrenamiento, el número de ciclos consecutivos sin mejorar el *loss* de validación para parar el proceso o el número de veces que se entrena cada RNA.
            - ¿Por qué no se debe usar una función de transferencia lineal en las neuronas de las capas ocultas?
            - El resto de modelos no tienen como parámetro el número de veces que se entrena, ¿por qué? Si se entrenan varias veces, ¿qué propiedades estadísticas tendrán los resultados de estos entrenamientos?
        - SVM: kernel y C, junto con los parámetros propios de cada kernel.

        - Árboles de decisión: profundidad máxima del árbol.
        - kNN: valor de k (número de vecinos a considerar).
    - Matrices de entradas y salidas deseadas ya normalizadas.
        - Como se ha puesto anteriormente, las salidas deseadas deberán indicarse como un vector donde cada elemento es la etiqueta correspondiente a cada patrón (por tanto, de tipo *Array{Any,1}*). En el caso de entrenar RR.NN.AA., las salidas deseadas deberán ser codificadas como se ha realizado en prácticas anteriores.
            - ¿Ha sido necesario normalizar las salidas deseadas? ¿Por qué?
        - Como se ha descrito previamente, en el caso de usar técnicas como SVM, árboles de decisión o kNN, 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 dos vectores (salidas y salidas deseadas) de tipo *Array{Any,1}*.
    - Índices de validación cruzada. 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 realizar validación cruzada con los mismos datos y las mismas particiones en todos los casos.

- Dado que la inmensa mayoría del código va a ser el mismo, no desarrollar 4 funciones distintas, una para cada modelo sino una única función, que debe ser la misma para los 4 modelos. En su interior, en cada *fold*, a la hora de generar el modelo, y dependiendo del mismo, se realizarán los siguientes cambios:
    - Si el modelo es una RNA, se codificarán las salidas deseadas y se usará el código desarrollado al respecto en prácticas anteriores. Como este modelo es no determinístico, será necesario realizar un nuevo bucle para entrenar varias RR.NN.AA. en el que se dividan los datos de entrenamiento en entrenamiento y validación (si se utiliza conjunto de validación) y se llame a la función

definida en prácticas anteriores para crear y entrenar una RNA.

  - Si el modelo no es una RNA, se desarrollará el código que entrene el modelo. Este código deberá ser la misma para cada uno de los 3 tipos restantes de modelos (SVM, Árboles de Decisión, kNN), con la modificación de que la línea en la que se cree el modelo debe ser distinta para cada uno de los modelos.

- Por su parte, esta función debería devolver, como mínimo, los resultados de las métricas seleccionadas.

- Una vez desarrollada esta función, comienza la parte experimental de las prácticas. El objetivo es determinar qué modelo con una combinación concreta de hiperparámetros ofrece mejores resultados, para lo cual se ejecutará la función anterior para cada uno de los 4 tipos de modelos, y para cada modelo se ejecutará con distintos valores en sus hiperparámetros.

  - Los resultados que se obtengan deberán ser documentados en la memoria a realizar, para lo cual será útil mostrar los resultados en forma de tabla y/o gráfica.

  - A la hora de mostrar una matriz de confusión en la memoria, una cuestión importante es cuál utilizar, habiendo realizado tantos entrenamientos. 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 parámetros 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:

En esta práctica, a la función 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 un SVM, se podría crear una variable de la siguiente manera:

```
parameters = Dict("kernel" => "rbf", "degree" => 3, "gamma" => 2, "C" => 1);
```

Otra forma de definir esa variable podría ser la siguiente:

```
parameters = Dict();

julia> parameters["kernel"] = "rbf";

julia> parameters["kernelDegree"] = 3;

julia> parameters["kernelGamma"] = 2;

julia> parameters["C"] = 1;
```

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

```
model = SVC(kernel=parameters["kernel"], degree=parameters["kernelDegree"], gamma=parameters["kernelGamma"], C=parameters["C"]);
```

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

Otro tipo de Julia que puede ser interesante 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 esta práctica, se puede utilizar para indicar qué modelo se quiere entrenar, por ejemplo *:ANN*, *:SVM*, *:DecisionTree* o *:kNN*.

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