La librería Scikit-Learn

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 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, 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 librería Scikit-Learn hay 3 implementaciones de SVM, llamadas SVC, NuSVC, y LinearSVC. Para realizar clasificación multiclase, dos de ellas utilizan la estrategia "one-vs-one" y la otra "one-vs-all". Consultad en la documentación de la librería qué estrategia utiliza cada una. Si bien SVC y NuSVC podrían ser utilizadas en estas prácticas, LinearSVC tiene una limitación importante que lo impide, ¿cuál es?

En la documentación de la librería se puede encontrar una explicación acerca de los parámetros que aceptan cada una de estas funciones. Las direcciones para consultar estos hiperparámetros son las siguientes:

- https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
- https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
- https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

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:

- Desarrollar una función llamada *modelCrossValidation* para que, además de entrenar redes de neuronas, también se realice validación cruzada sobre SVM, árboles de decisión y kNN. 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. Para entrenar SVM, árboles de decisión y kNN, se utilizarán símbolos con el mismo nombre que el modelo usado en Scikit-Learn, es decir, los siguientes símbolos: *:SVC*, *:DecisionTreeClassifier* y *:KNeighborsClassifier*.

    - El segundo, *modelHyperparameters*, contiene un objeto de tipo *Dict* con los hiperparámetros del modelo. Para el caso de los hiperparámetros de redes de neuronas, descritos en el ejercicio anterior, 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*.

      En el caso de los hiperparámetros de los modelos utilizados de la librería Scikit-Learn, los hiperparámetros deberán tener el mismo nombre que los indicados en la documentación de la librería, que se pueden consultar en las siguientes direcciones, una para cada modelo:

        - https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
        - https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
        - https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Dada la gran cantidad de hiperparámetros que se pueden especificar de acuerdo con la documentación de la librería, en este ejercicio solamente será necesario fijar unos pocos valores comunes para cada modelo. De esta manera, para cada modelo se deberán definir los siguientes hiperparámetros:

- SVM (*:SVC*): *C*, *kernel*, *degree*, *gamma* y *coef0*. El hiperparámetro *kernel* puede tomar los valores "linear", "poly", "rbf" y "sigmoid". 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*. Tanto en el enlace de documentación como en los apuntes de teoría se pueden ver los parámetros usados por cada kernel. El hiperparámetro C se usa en todos los casos.

- Árboles de decisión (*:DecisionTreeClassifier*): *max_depth*, que contiene la profundidad máxima de los árboles.

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

- *inputs*, de tipo *AbstractArray{<:Real,2}*

- *targets*, de tipo *AbstractArray{<:Any,1}*

- *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 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. 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, una modificación que puede ser importante es, en los modelos de la librería Scikit-Learn, antes de entrenar ningún modelo, transformar el vector de salidas deseadas en un vector de cadenas de texto, para evitar cualquier posible error con la librería de Python. Esto se puede hacer, sencillamente, con la siguiente línea:

```
targets = string.(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, entrenarlo, aplicarle el conjunto de test, calcular las métricas con la función confusionMatrix, y asignar estos valores a las posiciones correspondientes de los vectores. Este código a desarrollar deberá ser el mismo para cada uno de los 3 tipos de modelos (SVM, Árboles de Decisión, kNN), con la salvedad de que la línea en la que se cree el modelo debe ser distinta para cada uno de los modelos. Además, 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 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}*.

Esta función deberá devolver lo mismo que la realizada en el ejercicio anterior: una tupla con 7 valores, uno para cada métrica, donde cada valor, 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 7 valores deberán ser, por este orden: precisión, tasa de error, sensibilidad, especificidad, VPP, VPN y F1.

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* 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 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 la siguiente manera:

```
model = SVC(C=modelHyperparameters["C"], kernel=modelHyperparameters["kernel"],
degree=modelHyperparameters["degree"],     gamma=modelHyperparameters["gamma"],
coef0=modelHyperparameters["coef0"]);
```

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

Firmas de las funciones:

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

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