Validación cruzada

Aprendizaje Automático

Con el código desarrollado hasta el momento es posible entrenar una RNA y ofrecer una estimación de los resultados que ofrecería en su ejecución real (con patrones no vistos, representados por un conjunto de test). Sin embargo, en este último aspecto existen dos factores a tener en cuenta, como consecuencia del carácter no determinístico del proceso que estamos siguiendo:

- La partición del conjunto de patrones en entrenamiento/test es aleatoria (*hold out*), y por lo tanto es excesivamente dependiente de la buena o mala suerte al escoger los patrones de entrenamiento y test.

- El entrenamiento de la RNA no es determinístico, hay una parte que se basa en el azar, y se corresponde con la inicialización aleatoria de los pesos. Al igual que antes, es demasiado dependiente de la suerte o mala suerte de comenzar el entrenamiento en un buen o mal punto de partida.

Por estos dos motivos, el resultado en test de un único entrenamiento no resulta significativo a la hora de valorar la bondad del modelo ante patrones no vistos. Para solucionar este problema, cuando hay un factor dado por el azar, se repite el experimento varias veces y se promedian los resultados. Esto se puede implementar de una forma sencilla mediante un bucle; sin embargo, al tener dos fuentes de azar distintas, es necesario realizar esto de una forma ordenada.

En primer lugar, para minimizar el azar debido a la partición del conjunto de datos, es necesario disponer de un método que asegure que cada dato es usado para entrenamiento al menos una vez, y para test al menos una vez. El método más utilizado es el de validación cruzada (*cross-validation*). En este método, se parte el conjunto de datos en k subconjuntos disjuntos y se realizan k experimentos. En el k-ésimo experimento, el subconjunto k se separa para realizar test, y los k-1 restantes se utilizan para entrenar, realizando un *k-fold crossvalidation*. Un valor habitual suele ser k=10, con lo que se tiene un *10-fold crossvalidation*. Finalmente, el valor de test correspondiente a la métrica adecuada será el valor promedio de los valores de cada uno de los k experimentos.

Una variante de validación cruzada muy usada es la estratificada (*stratified cross-validation*). En ella, cada subconjunto se crea de tal forma que mantenga la proporción de patrones de cada clase igual (o similar) que en el *dataset* original. Esta es especialmente utilizada cuando el conjunto de datos está desbalanceado.

Es habitual guardar no solamente la media, sino también los k valores, para poder realizar posteriormente un contraste de hipótesis *pareado* con otro modelo. Para realizar esto, es necesario que ambos modelos hayan sido entrenados utilizando los mismos conjuntos de entrenamiento y test.

Se suele considerar que esta forma de evaluar el modelo es ligeramente pesimista, es decir, los resultados obtenidos en test son un poco peores que los que se obtendrían de un entrenamiento real con todos los datos disponibles. En un experimento *hold out*, como ya se ha dicho, se separan varios datos para realizar test. Esto hace que el modelo se entrene con menos datos de los disponibles, y que por azar los datos separados para test puedan tener una gran importancia (sobre todo si hay pocos datos). Por este motivo, al entrenar con menos datos y posiblemente sin datos "importantes", *hold out* se considera una evaluación pesimista. De la misma manera, *crossvalidation* separa también datos para test, con lo que no se entrena con todos los datos disponibles, por lo que también es pesimista. Sin embargo, se garantiza que todos los datos se usan al menos una vez en entrenamiento y una vez en test, con lo que se intenta minimizar el impacto del azar al separar datos, por lo que se considera una evaluación solamente ligeramente pesimista.

Realizar esto es tan sencillo como dividir el conjunto de datos y realizar un bucle con k iteraciones en el que en el ciclo k se entrene y evalúe un modelo con los conjuntos correspondientes. Sin embargo, si el modelo no es determinístico el resultado obtenido en el ciclo k no será significativo, puesto que es nuevamente dependiente del azar. En este caso, lo que hay que hacer es, dentro del ciclo k, un segundo bucle anidado en el que se entrene repetidamente el modelo, y finalmente se haga un promedio de los resultados para emitir finalmente el resultado del ciclo k. El número de entrenamientos debe de ser elevado para que el promedio de resultados sea realmente significativo, como mínimo unos 50 entrenamientos.

- Si se realiza este segundo bucle con un modelo determinístico, ¿cuál será la desviación típica de los resultados obtenidos en test? ¿Existe alguna diferencia entre realizar este segundo bucle y promediar los resultados, o hacer un único entrenamiento?

De esta manera, es posible evaluar un modelo junto con sus hiperparámetros en la resolución de un problema. Una situación muy común es querer comparar varios modelos (o el mismo modelo con distintos hiperparámetros), para lo cual lo que hay que hacer es aplicar este esquema con una salvedad importante: los conjuntos usados en la validación cruzada deben ser los mismos para cada modelo. Dado que la repartición de patrones en distintos conjuntos es aleatoria, el tener los mismos subconjuntos en distintas ejecuciones se consigue fijando la semilla aleatoria al principio del programa a ejecutar. Fijar la semilla aleatoria no solamente permite generar los mismos subconjuntos, sino también es importante para poder repetir los resultados en distintas ejecuciones.

Es importante tener en cuenta también que esta metodología permite estimar el rendimiento real de un modelo (aunque ligeramente pesimista). El modelo final que se utilizaría en producción sería el resultado de entrenarlo con todos los patrones disponibles, puesto que, como se ve en clase de teoría, y muy en líneas generales, con cuantos más patrones se entrene, mejor será el modelo.

Además, otra cuestión que suele ser de gran interés es obtener una matriz de confusión en test como resultado de hacer una validación cruzada. En general, esto no es posible, puesto que se entrenan una gran cantidad de redes de neuronas por cada *fold*. Si se entrenase un único modelo por cada *fold* (como se hace en el siguiente ejercicio), entonces se podría tener una matriz de confusión para cada uno de los conjuntos de test, y la matriz de confusión final sería la suma de estas matrices. Sin embargo, al tener muchas matrices de confusión por cada *fold*, aquí se utilizará una aproximación parecida: dentro de cada *fold* se calculará la matriz de confusión para cada ejecución, y se realizará el promedio de todas estas matrices. Finalmente, se sumarán estas matrices de confusión promedio de cada *fold* para desarrollar la matriz de confusión global en test.

Al hacer esta tarea para construir las matrices de confusión en test, es importante darse cuenta de varios factores:

- Esta forma de construir la matriz de confusión da como resultado una matriz que no es el resultado de un modelo concreto. Por lo tanto, llamarla “matriz de confusión” resulta un tanto engañoso, aunque resulta útil para ver la distribución de las instancias en test.

- Por este mismo motivo, las métricas que se puedan desarrollar a partir de esta matriz de confusión, en general no van a dar los mismos valores que el resto de valores de métricas devueltos por esta función (promedio de métricas para cada *fold*). Estas matrices de confusión serán unos promedios que se utilizarán para mostrar los resultados, y las métricas correctas serán los otros valores devueltos.

- Como consecuencia del cálculo de estas matrices de confusión promedio en cada *fold*, los valores de las matrices de confusión serán números reales. Si bien esto no es correcto, se pueden utilizar en el informe correspondiente, siempre que se explique la naturaleza de las matrices.

En este ejercicio, se pide:

- Desarrollar una función llamada *crossvalidation* que reciba un valor N (igual al número de patrones), y un valor de k (número de subconjuntos en los que se va a partir el conjunto de datos), y devuelva un vector de longitud N, donde cada elemento indica en qué subconjunto

debe ser incluido ese patrón.

Para hacer esta función, una posibilidad es realizando estos pasos:

1. Crear un vector con k elementos ordenados, desde 1 hasta k.
2. Crear un vector nuevo con repeticiones de este vector hasta que la longitud sea mayor o igual a N. Para hacer esto, consultar las funciones *repeat* y *ceil*.
3. De este vector, tomar los N primeros valores.
4. Utilizando la función *shuffle!*, desordenar este vector y devolverlo. Para usar esta función es necesario cargar el módulo *Random*.

Esta función debería devolver, por lo tanto, un vector de valores enteros, con una longitud igual a N. Para realizar esta función no se permite utilizar ningún bucle.

- Realizar una nueva función llamada *crossvalidation*, que en este caso reciba como primer argumento el vector *targets* de tipo *AbstractArray{Bool,1}* con las salidas deseadas, y como segundo argumento un valor de k (número de subconjuntos en los que se va a partir el conjunto de datos), y devuelva un vector de longitud N (N es igual al número de elementos de *targets*), donde cada elemento indica en qué subconjunto debe ser incluido ese patrón, y además esta partición sea estratificada. Para realizar esto, se pueden seguir los siguientes pasos:

  1. Crear un vector de índices, con tantos valores como filas en la matriz *targets*.
  2. Hacer una llamada a a la función *crossvalidation* desarrollada anteriormente pasando como parámetros el número de instancias positivas y el valor de k.
  3. Asignar, en el vector de índices, solamente en las posiciones correspondientes a las instancias positivas, el resultado de la llamada anterior.
     - ¿Podrías hacer estas 2 operaciones en una sola línea? (Esto no es un requisito de la función)
  4. Hacer algo similar, pero para las instancias negativas.
  5. Devolver el vector de índices.

  En la realización de esta función no se debe utilizar ningún bucle. Además, esta función

debería devolver un vector de números enteros con una longitud igual a N.

- Realizar una nueva función llamada *crossvalidation*, que en este caso reciba como primer argumento *targets* de tipo *AbstractArray{Bool,2}* con las salidas deseadas, y como segundo argumento un valor de k (número de subconjuntos en los que se va a partir el conjunto de datos), y devuelva un vector de longitud N (N es igual al número de filas de *targets*), donde cada elemento indica en qué subconjunto debe ser incluido ese patrón, y además esta partición sea estratificada. Para realizar esto, se pueden seguir los siguientes pasos:

    1. Crear un vector de índices, con tantos valores como filas en la matriz *targets*.

    2. Hacer un bucle que itere sobre las clases (columnas de la matriz *targets*), y que realice lo siguiente:

        i. Tomar el número de elementos que pertenecen a esa clase. Esto se puede hacer haciendo una llamada a la función *sum* aplicada a la columna correspondiente.

        ii. Hacer una llamada a la función *crossvalidation* desarrollada anteriormente pasando como parámetros este número de elementos y el valor de k.

        iii. Asignar, dentro del vector de índices, en las posiciones indicadas por la columna correspondiente de la matriz *targets*, los valores del vector resultado de esta llamada a la función *crossvalidation*.

        - ¿Podrías hacer estas 3 operaciones en una sola línea? (Esto no es un requisito de la función)

    3. Devolver el vector de índices.

    Como se puede ver en esta explicación, para realizar esta función se puede hacer un único bucle que recorra las clases. Además, esta función debería devolver un vector de números enteros con una longitud igual a N.

    Importante: Tanto para esta función como para la anterior, es necesario asegurarse que cada clase tenga al menos k patrones. Un valor habitual es k=10. Por tanto, es importante asegurarse de que se tienen al menos 10 patrones de cada clase.

        - ¿Qué ocurriría si alguna clase tiene un número de patrones inferior a k? ¿Qué consecuencias tendría a la hora de calcular las métricas?

- Si, por el motivo que sea, fuese imposible asegurar que se tienen al menos k patrones de cada clase, una posibilidad sería bajar el valor de k. En este caso, consultar con el profesor para valorar esta opción, y qué impacto podría tener en el resultado final de los modelos entrenados.

- Realizar una última función llamada *crossvalidation*, pero en este caso con el primer parámetro *targets* de tipo *AbstractArray{<:Any,1}* (es decir, un vector con elementos heterogéneos), el mismo segundo argumento, y realice validación cruzada estratificada.

  Para hacer esta función simplemente es necesario hacer una llamada a la función *oneHotEncoding* pasando el vector *targets* como argumento, y posteriormente hacer una llamada a la función *crossvalidation* anterior.

  Esta función, al igual que las anteriores, debería devolver un vector de valores enteros con los índices. Para el desarrollo de esta función no se permite utilizar bucles.

- Desarrollar una función nueva, llamada *ANNCrossValidation*. Esta función deberá entrenar varias veces un modelo siguiente la estrategia de validación cruzada descrita en esta práctica. Esta función recibirá las entradas propias del entrenamiento de redes de neuronas vistas en la práctica anterior, y, adicionalmente, las necesarias para hacer una validación cruzada y entrenar varias veces en cada *fold*. La lista completa de parámetros es la siguiente:

  - *topology*, de tipo *AbstractArray{<:Int,1}.*

  - *dataset*, de tipo Tuple{AbstractArray{<:Real,2}, AbstractArray{<:Any,1}}, con una tupla que contiene el conjunto de datos. El primer valor será la matriz de entradas, y el segundo un vector con las salidas deseadas, de tipo categórico. Para una RNA, estas deberán ser convertidas en una matriz de valores booleanos mediante el uso de la función *oneHotEncoding* desarrollada en prácticas anteriores. Esto se hará en el interior de esta función.

  - *crossValidationIndices*, de tipo *Array{Int64,1}*, que contendrá los índices para hacer una validación cruzada como resultado de llamar a la función *crossvalidation*.

  - Parámetros opcionales de entrenamiento. En la firma de la función al final de este documento se pueden ver los valores por defecto.

    - *numExecutions*, con el número de entrenamientos que se realizarán dentro de cada iteración o *fold* de la validación cruzada.

- *transferFunctions*, con las funciones de activación o transferencia de la RNA.
    - ¿Tendría sentido usar una función de transferencia lineal en las neuronas de las capas ocultas?
- *maxEpochs*, con el número máximo de ciclos que se entrenará cada RNA.
- *minLoss*, con el valor mínimo a alcanzar.
- *learningRate*, con la tasa de aprendizaje.
- *validationRatio*, con el ratio de patrones que serán utilizados para realizar validación en cada entrenamiento de la RNA haciendo una parada temprana. Este parámetro puede ser igual a 0, en caso de que no se quiera hacer validación cruzada.
- *maxEpochsVal*, con el número máximo de ciclos sin mejorar el *loss* de validación para realizar una parada temprana.

Esta función deberá, en primer lugar, calcular el vector de clases que tendrá, mediante una llamada a

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

Con este vector de clases, realizar una operación de *one-hot-encoding* del vector de salidas deseadas por medio de la función desarrollada, pasándole el vector *classes* recién calculado.

- ¿Qué podría ocurrir si no se le pasa el vector *classes*?

En segundo lugar, es necesario calcular el número de *folds* que se desea hacer. Esto se puede hacer mediante una llamada a la función *maximum* pasándole como argumento el vector de índices de validación cruzada. Una vez se tiene, se crearán varios vectores, uno para cada métrica, para almacenar el resultado del entrenamiento de la RNA de esa métrica. Por lo tanto, se necesitan vectores para precisión, tasa de error, sensibilidad, especificidad, VPP, VPN y F1. Además, es necesario crear una matriz con valores reales, inicialmente iguales a 0, con el resultado de la matriz de confusión, que se irá completando a medida que transcurran las iteraciones de la validación cruzada. Estos pasos son comunes a cualquier modelo a probar.

Después, se crear un bucle en el que se hagan tantas iteraciones como *folds*. En cada

iteración, se realiza lo siguiente:

- Se crean las variables de entrada y salida deseada tanto para entrenamiento como para test a partir de las matrices de entrada y salida deseada, de los índices de validación cruzada, y del número de *fold* (iteración del bucle) en el que se esté. Es decir, se tienen que crear 4 variables.

- Dado que las RR.NN.AA. son sistemas no deterministas, los resultados de un único entrenamiento por *fold* pueden no ser representativos del rendimiento con el conjunto de datos de ese *fold*. Por ese motivo, se deberá repetir el entrenamiento dentro del *fold* un número alto de veces, especificado como el hiperparámetro *numExecutions*. El conjunto de pasos será el siguiente:

  - En primer lugar, crear nuevos vectores para los resultados de cada entrenamiento para cada métrica. Esto también incluye el crear un *Array* tridimensional, de dimensiones *numClasses* x *numClasses* x *numExecutions*, donde *numClasses* es el número de clases, para almacenar las matrices de confusión de test de cada ejecución.

  - Crear un bucle en el que se realicen tantas iteraciones como se haya indicado en el hiperparámetro *numExecutions*. Dentro de cada iteración del bucle, se entrena una RNA mediante una llamada a la función *trainClassANN* desarrollada en ejercicios anteriores, y después se evalúa con el conjunto de test mediante una llamada a la función *confusionMatrix* desarrollada en ejercicios anteriores. Las métricas que devuelve esta llamada a *confusionMatrix* se almacenan en los vectores correspondientes, así como la matriz de confusión que devuelve.

    Además, en el caso de entrenar RR.NN.AA., el conjunto de entrenamiento puede ser dividido en entrenamiento y validación, si el ratio de patrones a usar para el conjunto de validación es mayor que 0, pasado como hiperparámetro de nombre *validationRatio*. En este caso, el conjunto de entrenamiento se divide en entrenamiento y validación. Para realizar esto, usar la función *holdOut* desarrollada en una práctica anterior. Es necesario tener en cuenta que, al utilizar la función *holdOut*, el ratio de patrones que se separa no debe ser igual a *validationRatio*, puesto que este es un porcentaje de patrones del conjunto total de datos, y en este *fold* el conjunto es menor,

puesto que se han separado datos para hacer test. Se deja como ejercicio sencillo el calcular cuál es el ratio de patrones que se tiene que poner en la llamada a la función *holdOut*, que es similar al empleado en la función *holdOut* con tasa de valores para validación y test.

- Una vez realizados todos los entrenamientos de este *fold*, los valores correspondientes de las métricas de este *fold* serán las medias de los vectores de las métricas correspondientes calculados para todos los entrenamientos realizados en este *fold*. Igualmente, la matriz de confusión de este *fold* será la matriz promedio de todas las devueltas en este *fold*. Para calcular la matriz de confusión promedio, se puede utilizar la función *mean* utilizando *dims*=3. Tened en cuenta que esto realizar la media a través de la tercera dimensión (que es el número de ejecuciones), pero no la elimina, sino que queda con un elemento en esa dimensión (el resultado del promedio). Esta matriz de confusión promedio deberá ser sumada a la matriz de confusión de test global (con valores inicialmente a 0). Para hacer esta suma, habrá que tener en cuenta que la matriz de confusión promedio de este *fold* tiene 3 dimensiones, con 1 elemento en la tercera dimensión, para referenciarla, por lo que, en definitiva, es una matriz de dos dimensiones con una dimensión "extra" que no se utiliza.

Como resultado, se tendrán los vectores de las métricas con los valores de cada *fold*, así como la matriz de confusión en test. Finalmente, esta función debería devolver, para cada métrica, una tupla con dos valores, donde el primero es el valor medio del vector, y el segundo la desviación típica (se puede calcular con la función *std*), y la matriz de confusión en test. Por lo tanto, esta función debe devolver una tupla con 8 elementos, donde cada uno de los 7 primeros elementos se corresponderá con una métrica distinta, y será, a su vez, una tupla con dos valores: media y desviación típica de la correspondiente métrica. Estos 7 valores se corresponderán, por este orden, con las métricas precisión, tasa de error, sensibilidad, especificidad, VPP, VPN y F1. El octavo valor será la matriz de confusión global en test.

Esta función será llamada por la función a desarrollar en el ejercicio siguiente. A pesar de que hasta el momento se ha visto sólo cómo entrenar RR.NN.AA., en el próximo ejercicio se utilizarán otros modelos mediante el uso de otra librería (Scikit-Learn). Tened en cuenta que otros modelos de Aprendizaje Automático son determinísticos, por lo que no necesitan del

bucle interior (siempre que se entrenan con los mismos datos devuelven las mismas salidas), sino únicamente del bucle para cada *fold*.

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 crossvalidation(N::Int64, k::Int64)
function crossvalidation(targets::AbstractArray{Bool,1}, k::Int64)
function crossvalidation(targets::AbstractArray{Bool,2}, k::Int64)
function crossvalidation(targets::AbstractArray{<:Any,1}, k::Int64)

function ANNCrossValidation(topology::AbstractArray{<:Int,1},
       dataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{<:Any,1}},
       crossValidationIndices::Array{Int64,1};
       numExecutions::Int=50,
       transferFunctions::AbstractArray{<:Function,1}=fill(σ, length(topology)),
       maxEpochs::Int=1000, minLoss::Real=0.0, learningRate::Real=0.01,
       validationRatio::Real=0, maxEpochsVal::Int=20)
```