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.

En esta práctica, 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*.

  No realizar ningún bucle en el desarrollo de esta función.

- 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:
    a. 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.
    b. Hacer una llamada a la función *crossValidation* desarrollada anteriormente pasando como parámetros este número de elementos y el valor de k.
    c. 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?
3. Devolver el vector de índices.

Como se puede ver en esta explicación, para realizar esta función se puede hacer un bucle que recorra las clases.

Importante: Para realizar esto, 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 10 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.

  En este caso, no se especifican los pasos para realizar esta función. Sin embargo, son similares a la anterior. Una forma sencilla de hacerla consistiría en llamar a la función *oneHotEncoding* pasando el vector *targets* como argumento.

    - ¿Podrías desarrollar esta función sin llamar a *oneHotEncoding*?

- Integrar estas funciones en el código desarrollado hasta el momento y realizar dos funciones para entrenar RR.NN.AA. siguiendo la estrategia *stratified cross-validation*. Para realizar esto:

    - En primer lugar, es necesario fijar la semilla aleatoria para asegurar que los experimentos son repetibles. Esto se puede hacer con la función *seed!* del módulo *Random*.

    - Una vez cargados y codificados los datos, generar un vector de índices mediante la llamada a la función *crossvalidation*.

    - Crear una función llamada *trainClassANN*, que recibe como parámetros la topología, el conjunto de entrenamiento y los índices utilizados para hacer validación cruzada. De forma opcional, puede recibir el resto de parámetros utilizados en prácticas anteriores.

    - En el interior de esta función se hará lo siguiente:

        - Crear un vector con k elementos, que contendrá los resultados en test del proceso de validación cruzada con la métrica seleccionada. Si se quiere usar más de una métrica, crear un vector por métrica.

        - Hacer un bucle con k iteraciones (k *folds*) donde dentro de cada iteración a partir de las matrices de entradas y salidas deseadas, por medio del vector de índices resultado de la función anterior, se creen 4 matrices: entradas y salidas deseadas para entrenamiento y test. Como siempre, hacer este proceso de crear nuevas matrices sin bucles.

- Dentro de este bucle, añadir una llamada para generar el modelo con el conjunto de entrenamiento, y hacer test con el conjunto de test que corresponda según el valor de k. Esto se puede hacer mediante una llamada a la función *trainClassANN* desarrollada en prácticas previas, pasando como parámetros los conjuntos correspondientes.

- Para el caso de entrenar RR.NN.AA., como se indicó en la práctica anterior, el entrenamiento de RR.NN.AA. no es determinístico, con lo que, para cada iteración k de la validación cruzada, será necesario entrenar varias RR.NN.AA. y devolver el promedio de los resultados de test (con la métrica o métricas seleccionadas) para tener el valor de test correspondiente a ese k. El número de entrenamientos podrá ser pasado también como argumento en la llamada a esta función.

- 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. Para realizar esto, usar la función *holdOut* desarrollada en una práctica anterior. El ratio de patrones para usar en el conjunto de validación podrá ser usada como argumento en esta función.

- Una vez se ha entrenado el modelo (varias veces) en cada *fold*, tomar el resultado y rellenar el vector o vectores creados anteriormente (uno para cada métrica).

- Finalmente, ofrecer el resultado de promediar los valores de estos vectores para cada métrica junto con sus desviaciones típicas.

- Como resultado de esta llamada, se debería devolver, como mínimo, el valor de test en la(s) métrica(s) seleccionada(s). Si el modelo no es determinístico (como es el caso de las RR.NN.AA.), será el promedio de los resultados de varios entrenamientos.

- Una vez se tenga hecha esta función, desarrollar una segunda, del mismo nombre, y que acepte como salidas deseadas un vector en lugar de una matriz, al igual que en una práctica anterior, y su funcionamiento simplemente sea hacer una llamada a esta función recién desarrollada.

- Observaciones:

- A pesar de que hasta el momento se ha visto sólo cómo entrenar RR.NN.AA., en la próxima práctica se utilizarán otros modelos mediante el uso de otra librería (Scikit-Learn). La idea es utilizar el mismo código que se usa para hacer validación cruzada con este bucle global, cambiando únicamente la línea en la que se genera el modelo.

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

function trainClassANN(topology::AbstractArray{<:Int,1},
    trainingDataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}},
    kFoldIndices::     Array{Int64,1};
    transferFunctions::AbstractArray{<:Function,1}=fill(σ, length(topology)),
    maxEpochs::Int=1000, minLoss::Real=0.0, learningRate::Real=0.01,
    numRepetitionsANNTraining::Int=1, validationRatio::Real=0.0,
    maxEpochsVal::Int=20)
function trainClassANN(topology::AbstractArray{<:Int,1},
    trainingDataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}},
    kFoldIndices::     Array{Int64,1};
    transferFunctions::AbstractArray{<:Function,1}=fill(σ, length(topology)),
    maxEpochs::Int=1000, minLoss::Real=0.0, learningRate::Real=0.01,
    numRepetitionsANNTraining::Int=1, validationRatio::Real=0.0,
    maxEpochsVal::Int=20)
```