Clasificación multiclase

Aprendizaje Automático

A la hora de resolver un problema de clasificación, muchos de los modelos de aprendizaje automático existentes permiten solamente separar dos clases, que se suelen denominar “positivo” y “negativo”. Los patrones “positivos” suelen ser los relacionados con aquello que se quiere detectar, como enfermedad, alarma, o un tipo de objeto en una imagen. Los patrones “negativos” suelen caracterizarse por la ausencia de esta característica que tienen los positivos. Para desarrollar una RNA que clasifique en dos clases, se necesita una única neurona en la capa de salida, con función de transferencia sigmoidal logarítmica (u otra similar), de tal forma que la salida de la RNA estará entre 0 y 1, y puede interpretarse como la certeza que tiene la RNA en clasificar un patrón como “positivo”. La clasificación en “negativo” o “positivo” se realiza de una forma sencilla, aplicando un umbral que suele estar en 0.5, aunque se puede cambiar.

Sin embargo, en muchas ocasiones se desea desarrollar un sistema que sea capaz de clasificar en más de dos clases. Un ejemplo sencillo es un sistema que quiera clasificar una imagen según si es un perro, gato o ratón, u otro tipo de animal. En este caso, se desea desarrollar un sistema de clasificación en 4 clases: “perro”/”gato”/”ratón”/”otro”. Si se desea entrenar una RNA para distinguir entre estos 3 animales, se necesita una neurona de salida para cada clase, incluyendo la clase “otro” (4 neuronas de salida en total).

En este esquema, como se ha realizado en prácticas anteriores, cuando hay más de dos clases generalmente se usa una codificación llamada en inglés *one-hot-encoding*, que se basa en, para cada patrón, crear un valor booleano con un valor correspondiente a cada clase, de tal forma que un cada valor booleano será igual a 1 si ese patrón pertenece a esa clase, y 0 en caso contrario. Al entrenar una RNA con este esquema, cada neurona de salida puede entenderse como un modelo especializado en clasificar en una clase determinada. En este tipo de redes se suele usar una función de transferencia lineal en la capa de salida, con lo que salidas negativas indican que esa neurona no clasifica el patrón en esa clase (es decir, desde el punto de vista de esa clase lo clasifica como “negativo”), y salidas positivas indican que una neurona clasifica el patrón como esa clase (es decir, desde el punto de vista de esa clase lo clasifica como “positivo”). El valor absoluto de la salida de una neurona indica la seguridad de esa neurona en la clasificación. Finalmente, la función *softmax* recibe esos valores de clasificación y los transforma de tal manera que estén entre 0 y 1, y sumen 1, interpretándose como la probabilidad de pertenencia a cada clase. El patrón se clasificará en aquella clase cuyo valor de salida sea más alto. La función *softmax* se define de la siguiente manera:

$$softmax(y_i) = \frac{e^{y^i}}{\sum_j e^{y^j}}$$

donde y^i es la salida de la neurona i. Por ejemplo, en un problema de clasificación de 3 clases si se tiene salidas de las 3 neuronas de [2, 1, 0.2], en este caso las 3 neuronas clasificarían las entradas como pertenecientes a sus respectivas clases, aunque la primera con mucha mayor certidumbre. Al pasar la función *softmax*, las probabilidades respectivas serán [0.65, 0.24, 0.11], con lo que se clasificará como la primera clase.

De esta forma, la función *softmax* convierte los valores reales que emiten las salidas de las neuronas de salida en valores de probabilidad, haciendo que cuanto más negativo sea un valor (más certeza de no pertenencia a esa clase), este se transforme en un valor más próximo a 0, y cuanto más positivo sea un valor (más certeza de pertenencia a esa clase), este se transforme en un valor más próximo a 1. Como se indicó antes, la suma de valores o probabilidades de la última capa será igual a 1. Por este hecho, es necesaria una cuarta clase especial "otro" en el ejemplo anterior, y en general una clase extra en el caso de que, con varias clases, un patrón pueda no pertenecer a ninguna de ellas.

- ¿Por qué es necesaria esta clase extra al usar la función *softmax*?
    - Ayuda: escribir en Julia *softmax([-1, -1, -0.2])*, e interpretar las entradas (¿qué representa el vector [-1, -1, -0.2] y cómo se interpreta?) y las salidas de la función (¿cuánto suman los valores? ¿qué dice cada una?)
        - Para utilizar esa función, importarla de la librería *Flux*.
- ¿Podría no ser necesaria crear la clase adicional? ¿Qué modificación habría que hacer en la RNA? ¿Cómo se interpretaría la salida? ¿Cómo se generaría la clase de salida en función de las salidas de las neuronas de salida?
- En general, ¿cómo tiene que ser la salida de un modelo para que no se necesite esta cuarta clase?
- ¿El modelo kNN necesita esta cuarta clase?
- ¿Cuántas clases serían necesarias si una RNA quisiera reconocer esos 3 tipos de animales, y, si no es uno de ellos, decir si es un animal o no? ¿Y si el modelo es un kNN?

Por lo tanto, al tener más de dos clases, ya no se aplica el esquema "positivo"/"negativo". El problema en estos casos es que muchos de los modelos de aprendizaje automático solamente son

capaces de separar dos clases, con lo que, en principio, no podrían utilizarse. Un ejemplo de este tipo de sistemas son las Máquinas de Vectores de Soporte (SVM), que se ven con más profundidad en clase de teoría. Sobre este modelo se han realizado modificaciones en su formulación para permitir clasificaciones multiclase; sin embargo, en la práctica no suelen usarse, y en su lugar se suele emplear una estrategia que permite usar las SVM binarias para clasificar en varias clases. Otro sistema que, por ahora, solamente permite separar en dos clases es el algoritmo DoME, que devuelve como modelo una ecuación matemática. Esta ecuación, al ser evaluada en una instancia, devuelve un valor real. Su signo indica si esta instancia será clasificada como positiva o negativa. Además, cuanto mayor valor absoluto tenga este valor, mayor será la certidumbre de clasificación en positivo o negativo.

Existen dos grandes estrategias para "convertir" problemas multiclase en problemas de clasificación binaria. Estas estrategias se llaman "uno contra uno" o "uno contra todos". En clase de teoría se explican ambas, pero dado que "uno contra todos" es mucho más utilizado, en prácticas se utilizará esta estrategia.

La estrategia "uno contra todos" se basa en, si se tiene un problema de clasificación de L clases, generar L problemas de clasificación binaria, uno por clase. En el l-ésimo problema, la clase l debe ser separada del resto, es decir, los patrones que pertenezcan a esa clase serán considerados "positivos", y los que no pertenezcan a ella serán considerados "negativos". Siguiendo con el ejemplo anterior de los animales, habría que resolver 3 problemas de clasificación distintos: uno para clasificar "perro"/"no perro", otro para clasificar "gato"/"no gato", y otro para clasificar "ratón"/"no ratón". Se entrenarían, por tanto, 3 clasificadores con las mismas entradas pero con salidas deseadas distintas para cada problema.

- En la descripción anterior, para estos 3 animales se utilizaron 4 clases, incluyendo la clase "otro". ¿Por qué no se entrena un clasificador para esta clase?

Una vez entrenados, para evaluar un nuevo patrón este se aplica en los 3 clasificadores y, en función de la salida, se toma una decisión. Si sólo uno de los sistemas tiene salida positiva, o ninguno de los tres lo clasifica como positivo, la decisión está clara. Sin embargo, en alguna ocasión más de un clasificador dará una salida positiva para un mismo patrón. Por suerte, muchos clasificadores, además de emitir una salida en forma de clasificación (en este caso "positivo"/"negativo"), dan información acerca del nivel de certidumbre o seguridad que tienen de que ese patrón sea clasificado como "positivo". Si más de un clasificador clasifica al patrón como positivo, se clasifica en la clase correspondiente al clasificador que tenga una mayor seguridad en su clasificación.

- ¿Se podrían usar las salidas de estos 3 clasificadores como entrada a la función *softmax*? ¿Qué consecuencias tendría?

- En general, cuando hay L clases y la posibilidad de que un patrón no pertenezca a ninguna de ellas, ¿cuál es el impacto de usar la función *softmax* en las salidas? ¿En qué casos se podría usar? ¿Por qué?

- La función *softmax* es útil para conseguir un valor de *loss* que permita entrenar la RNA. Sin embargo, si no se usara, en el ejemplo anterior de los animales, ¿sería necesaria la cuarta clase "otro"?

Finalmente, es necesario tener en cuenta otra posibilidad distinta a la hora de asignar los patrones a las clases. Hasta el momento, y en la mayoría de las situaciones, las clases consideradas son mutuamente excluyentes, es decir, en el ejemplo anterior, un animal o bien es perro, o bien es gato, o bien es ratón, o bien ninguna de las 3, pero no puede ser de varias clases a la vez. Este es el caso más común, pero en ocasiones algún problema tendrá clases que no son mutuamente excluyentes. Por ejemplo, al clasificar sonidos de animales según el animal que los emita, puede ocurrir que en un sonido se mezclen varios animales. En estos casos, el esquema utilizado de usar una función de transferencia lineal en la última capa junto con la función *softmax* no funcionaría, puesto que, de forma natural, la suma de las probabilidades de pertenencia a las clases puede ser mayor que 1 (puede pertenecer a varias clases a la vez). Para estos casos, el esquema que puede utilizarse para entrenar RR.NN.AA. es usar funciones de transferencia sigmoidales logarítmicas en la última capa (en lugar de lineales), que dan una salida entre 0 y 1, y no realizar transformación mediante la función *softmax*. De esta manera, la salida final de cada neurona de salida es independiente del resto de neuronas de salida, y más de una puede tomar valores cercanos a 1. La salida de cada neurona nuevamente se interpretaría como la probabilidad de pertenencia a esa clase, pero en este caso la suma de las probabilidades no tiene que ser 1 (son independientes). El no aplicar la función *softmax* tiene dos ventajas: la primera, ya mencionada, es que permite la clasificación en clases no mutuamente excluyentes; la segunda es que al no usar esta función ya no se necesita una clase adicional ("otro" en el ejemplo anterior) para los casos en que un conjunto de entradas pueda no pertenecer a alguna de las clases dadas.

- ¿Por qué ya no se necesita esta clase extra?

Ante un conjunto de entradas, como siempre, este se clasifica en la clase cuyo clasificador o neurona de salida (en caso de usar una RNA) haya mostrado una mayor seguridad. Este esquema de salidas no mutuamente excluyentes es similar al esquema "uno contra todos", en el que se entrenan en

paralelo un clasificador por clase. Los clasificadores son independientes y la clase final es la de aquel clasificador que tenga una mayor certeza de pertenencia a esa clase. Si todos los clasificadores devuelven como clasificación “negativo” y no existe la posibilidad de no pertenencia a ninguna clase, se clasifica en la clase correspondiente al clasificador que tenga la menor certeza de que es negativo. Si todos los clasificadores devuelven como clasificación “negativo” y sí existe la posibilidad de no pertenencia a ninguna clase, sencillamente se clasifica como “otro”.

La siguiente tabla muestra un resumen de las situaciones a la hora de usar una RNA para resolver un problema de clasificación. Tened en cuenta que en el caso de clasificación binaria no se contempla la posibilidad de que un conjunto de entradas no pertenezcan a ninguna clase, puesto que en este caso se estaría en clasificación multiclase.

		¿Se añade clase extra si puede no pertenecerse a ninguna de las dadas?	Función de transferencia capa de salida	Intervalo salida de la neurona de salida	Función que se aplica a las salidas (capa adicional)	Intervalo salida final	¿En qué clase se clasifica?
Clasificación binaria		-	sigmoidal logarítmica	[0, 1]	-	[0, 1]	“negativo”/”positivo” según la salida mayor o igual a un umbral (0.5 para salidas en [0, 1])
Multiclase	Clases mutuamente excluyentes	Sí	lineal	[-inf, inf]	*softmax*	[0, 1]	Clase cuya salida tiene el valor más próximo a 1
	Clases no mutuamente excluyentes	No	sigmoidal logarítmica	[0, 1]	-	[0, 1]	Clases con salida mayor o igual a un umbral (0.5 para salidas en [0, 1]) (podrían ser varias, o ninguna)

En el caso de usar una estrategia “uno contra todos”, esta sería similar a la última fila, excepto por el hecho de que el intervalo no sería necesariamente [0, 1], sino que estaría condicionado por el modelo que se usase, y, por lo tanto, el umbral también. Por ejemplo, un SVM emite salidas entre $-\infty$ y $+\infty$, con lo que el umbral típico está en el 0.

Otro factor a tener en cuenta al tratar problemas multiclase es la métrica escogida. La mayoría de las métricas estudiadas (VPP, sensibilidad, etc.) corresponden a problemas de clasificación binaria. Cuando el número de clases es superior a 2, estas métricas se siguen pudiendo usar; sin embargo, su uso es ligeramente distinto.

Cuando el número de clases es superior a dos, las métricas VPP, VPN, sensibilidad y especificidad se pueden calcular de forma separada para cada clase. De esta forma, para una clase concreta los

patrones positivos son los clasificados en esa clase, y los negativos son los clasificados en cualquiera de las otras clases. Por lo tanto, desde el punto de visto exclusivo de esa clase, se puede calcular VP, VN, FP y FN, y a partir de ellos los valores de sensibilidad, especificidad, VPP y VPN para esa clase en concreto, y, finalmente, el valor de *F_1-score*. Esta forma de tratar las clases de forma separada es parecida al desarrollo de varios clasificadores en la estrategia "uno contra todos" (en el caso de entrenar clasificadores binarios que no permitan realizar clasificación multiclase). Una vez calculadas estos valores, estos se pueden combinar en uno único que será el usado para valorar el rendimiento del clasificador. Para esto, existen 3 estrategias: *macro*, *weighted* y *micro*, de las cuales en prácticas usaremos solamente las dos primeras:

- *Macro*. En esta estrategia, se calculan las métricas como sensibilidad, VPP o *F_1-score* como una media aritmética de las métricas correspondientes a cada clase. Al ser media aritmética, no tiene en cuenta el posible desbalanceo entre clases.
- *Weighted*. En esta estrategia, se calculan las métricas como sensibilidad, VPP o *F_1-score* como una media de las métricas correspondientes a cada clase ponderado por el número de patrones que pertenecen (salida deseada) a cada clase. Por lo tanto, es adecuada cuando las clases están desbalanceadas.
- *Micro*. Se calcula el número de VP, FN y FP de forma global. Cuando las clases son mutuamente excluyentes, el resultado tanto de micro-sensibilidad, micro-VPP o *micro-F_1-score* es idéntico al de la precisión. Por lo tanto, esta métrica es de utilidad cuando existen clases no mutuamente excluyentes.

En este ejercicio, se pide:

- Desarrollar una función llamada *confusionMatrix* (el mismo nombre que en la práctica anterior) que permita devolver los valores de las métricas adaptadas a la condición de tener más de dos clases. Para ello, incluir un parámetro adicional que permita calcularlas de las formas *macro* y *weighted*.

  Esta función debería recibir dos matrices: salidas del modelo (*outputs*) y salidas deseadas (*targets*), ambas de elementos booleanos y dimensión 2, de tipo *AbstractArray{Bool,2}*, con cada patrón en una fila y cada clase en una columna. Lo primero que debería hacer esta función es comprobar que el número de columnas de ambas matrices es igual y es distinto de 2. Para el caso de que tengan una sola columna, se toman esas columnas como vectores y se llama a la función *confusionMatrix* desarrollada en la práctica anterior.

- ¿Por qué no son válidas las matrices de dos columnas?

Si ambas matrices tienen más de dos columnas, se pueden ejecutar los siguientes pasos:

- Reservar memoria para los vectores de sensibilidad, especificidad, VPP, VPN y F_1, con un valor por clase, inicialmente iguales a 0. Para realizar esto, se puede usar la función *zeros*.

- Iterar para cada clase, y realizar una llamada a la función *confusionMatrix* de la práctica anterior pasando como vectores las columnas correspondientes a la clase de esa iteración de las matrices *outputs* y *targets*, como vectores. Asignar el resultado en el elemento correspondiente de los vectores de sensibilidad, especificidad, VPP, VPN y F_1.

- Reservar memoria para la matriz de confusión, y hacer un bucle doble en el que ambos bucles iteren sobre las clases, para ir rellenando todas las celdas de la matriz de confusión. Los valores de las celdas de la matriz de confusión deben seguir la distribución de las clases de teoría.

    - En realidad no es necesario reservar memoria y hacer un bucle doble. Esto se puede hacer de una forma más elegante y eficiente mediante una *comprehension*, en una sola línea de código, sin necesidad de reservar la memoria. Se deja como ejercicio voluntario el realizarlo. Como ayuda, puede ser útil escribir el bucle doble con una sola línea en el cuerpo del bucle, y posteriormente intentar convertirlo en una *comprehension*.

- Unir los valores de sensibilidad, especificidad, VPP, VPN y F_1 para cada clase que se tienen guardados en sus respectivos vectores en un único valor usando la estrategia *macro* o *weighted* según se haya especificado en el argumento de entrada.

    - Cuando se usa la estrategia *weighted*, es necesario calcular cuántas instancias pertenecen a cada clase para hacer la media ponderada. Esto se puede hacer mediante *sum(targets, dims=1)*. Sin embargo, esta llamada devuelve una matriz con una fila (y tantas columnas como clases). Si se hace el producto elemento a elemento de este resultado por uno de los vectores a promediar (por ejemplo, el de sensibilidades), el resultado será una matriz, puesto que el vector de sensibilidades se interpreta como una columna que, multiplicado por una matriz con una fila, genera una matriz. Esto se puede solucionar de

varias maneras, siendo la más sencilla de todas convertir la matriz con el número de instancias en un vector, haciendo algo similar a *vec(sum(targets, dims=1))*.

- Finalmente, calcular el valor de precisión con la función *accuracy* desarrollada en una práctica anterior, y a partir de este valor calcular la tasa de error.

Esta función debería devolver una tupla con los mismos valores especificados en la práctica anterior, por el mismo orden (precisión, tasa de fallo, sensibilidad, especificidad, VPP, VPN, F1, matriz de confusión). Además, como se puede ver, para hacer esta función se permite utilizar bucles.

- Desarrollar otra función llamada *confusionMatrix* en la que el primer parámetro *outputs* sea de tipo *AbstractArray{<:Real,2}*, y targets sea de tipo *AbstractArray{Bool,2}* (el mismo que antes). Lo que debería hacer esta función es convertir el primer parámetro a una matriz de valores booleanos (mediante la función *classifyOutputs*) y llamar a la función anterior, devolviendo por tanto los mismos valores. Además, esta función puede recibir dos valores opcionales: *threshold* y *weighted*, que se usarán para hacer las llamadas respectivas a las funciones en las que se basa esta.

  - Dentro de esta función, ¿en qué llamadas se usará cada uno de estos dos parámetros opcionales?

  - ¿En qué casos es necesario el uso del parámetro *threshold*?

  Dado que se basa en la función anterior, esta no debe contener bucles.

- Sobrecargar esta función una vez más desarrollando otra del mismo nombre que realice la misma tarea, pero esta vez tome como entradas tres vectores (*outputs*, *targets* y *classes*) cuyos elementos sean de cualquier tipo (es decir, que sean de tipo *AbstractArray{<:Any}*), además del parámetro opcional que permita calcular las métricas de las formas *macro* y *weighted*. Los vectores *outputs* y *targets* tendrán las etiquetas de salida obtenidas y deseadas para cada instancia, y, por lo tanto, deberán tener de la misma longitud. El vector *classes* tendrá todas las posibles clases, y, por lo tanto, tendrá una longitud distinta a estos dos. Los elementos de estos vectores representan las etiquetas, y pueden estar de distintas formas, como valores reales, enteros, cadenas de caracteres, símbolos, etc. Por este motivo, se utiliza el tipo *Any*. Por ejemplo, las clases pueden ser *[“perro”, “gato”, 3, :gree]*.

Como es obvio, es necesario que tanto las etiquetas obtenidas (vector *outputs*) como las salidas deseadas (vector *targets*) estén incluidas en el vector *classes*.

- Escribir esta línea sin ningún tipo de bucle. Para ello, puede ser útil consultar las funciones *all*, *in* y *unique*. Al final de esta práctica se da la solución de cómo realizar esta línea.

Esta función se puede hacer mediante una llamada a una versión anterior de la función *confusionMatrix*, concretamente la que recibe matrices (de dos dimensiones) de valores booleanos, además del valor opcional. Para realizar esta llamada, se codificarán ambas matrices, *outputs* y *targets*, mediante una llamada a la función *oneHotEncoding* pasando como argumento el vector *classes*. Con el resultado de estas dos codificaciones, se puede llamar a la función *confusionMatrix*.

- Es importante tener el vector *classes* de forma previa a las llamadas a *oneHotEncoding* y se pase el mismo en ambas llamadas de codificación. ¿Qué podría ocurrir si no se hace de esta manera?

Dado que se basa en realizar una llamada a la función anterior, esta función debería devolver los mismos valores, y no se permite el uso de bucles.

- Realizar una última versión de *confusionMatrix*, similar a la anterior, pero sin recibir el vector *classes*, sino que las clases se calculen a partir de las etiquetas de salidas y salidas deseadas. Para hacer esto, es necesario construir el vector *classes* a partir de *targets* y *outputs* mediante la función *unique*, de la siguiente manera:

```
classes = unique(vcat(targets, outputs));
```

Una vez hecho esto, se hará una llamada a la versión anterior de esta función, pasando los 3 vectores (*outputs*, *targets* y *classes*), así como el parámetro opcional.

Esta función puede ser útil en el caso de que el usuario no desee especificar el vector de clases, puesto que lo considere redundante al tener el vector de salidas deseadas o de salidas obtenidas. Sin embargo, ha de utilizarse con cuidado, puesto que es posible, al particionar los datos, que alguna clase no esté representada en algunos de estos vectores.

Dado que se basa en realizar una llamada a la función anterior, esta función debería devolver los mismos valores, y no se permite el uso de bucles.

- Al igual que en la práctica anterior, crear cuatro funciones llamadas *printConfusionMatrix*, en este caso para el caso de tener más de dos clases, y, por lo tanto, tanto *outputs* como *targets* serán matrices bidimensionales. En una, *outputs* será de tipo *AbstractArray{Bool,2}*, en otra será de tipo *AbstractArray{<:Real,2}*, con *targets* de tipo *AbstractArray{Bool,2}* en estos dos primeros casos. En los otros dos casos, tanto tanto *outputs* como *targets* serán de tipo *AbstractArray{<:Any,1}*., y de tipo *AbstractArray{<:Any,1}*. Una de ellas recibirá el vector *classes*, y otra lo calculará a partir de *targets* y *outputs* al igual que las funciones *confusionMatrix* respectivas. Además, estas funciones recibirán el parámetro para calcular las métricas de la forma *macro* o *weighted*. Estas funciones no serán evaluadas; sin embargo, serán de utilidad para realizar la práctica 2.

- Crear una función llamada *trainClassDoME*, que ejecute el algoritmo DoME para obtener modelos matemáticos a partir de un *dataset*, para resolver un problema de clasificación en dos clases. Esta función recibe como parámetros:

    - *trainingDataset*, de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,1}}*, con una tupla con el conjunto de entrenamiento. El segundo valor de la tupla será un vector de valores booleanos, y, por lo tanto, el problema de clasificación será binario.

    - *testInputs*, de tipo *AbstractArray{<:Real,2}*, con las entradas del conjunto de test a las que aplicar los modelos obtenidos.

    - *maximumNodes*, de tipo *Int*, con el número máximo de nodos que puede tener la expresión generada por el algoritmo. Valores típicos oscilan entre 20 y 40, pero pueden ser superiores para problemas más complejos.

  Para poder ejecutar el algoritmo DoME, será necesario tener instalado el paquete *SymDoME*, y posteriormente importarlo con *using SymDoME*.

  Esta función comenzará convirtiendo las entradas tanto del conjunto de entrenamiento como del conjunto de test en matrices con valores de tipo *Float64*. Este algoritmo se basa en el uso de ecuaciones para calcular valores con gran precisión. Por este motivo, es de interés que el tipo de los datos sea *Float64* en lugar de *Float32*.

  Para entrenar, se dispone de la función *dome*, que recibe como parámetros obligatorios la matriz de entradas y el vector de salidas deseadas. Si se desea resolver un problema de clasificación, los elementos de este vector serán valores booleanos. Como argumento opcional, entre otros, recibe el número máximo de nodos que tendrá el árbol que representa

la expresión matemática, con el nombre *maximumNodes*. Esta función *dome* devuelve una tupla con 4 valores, siendo el último de ellos el modelo matemático obtenido, que es el que interesa aquí. Por tanto, la llamada a esta función será la siguiente:

```
_, _, _, model = dome(trainingInputs, trainingTargets;
maximumNodes = maximumNodes)
```

Una vez se tiene el modelo, este se puede convertir a una cadena de caracteres para examinarlo llamando a la función *string(model)*. Incluso se puede convertir a una cadena de caracteres en formato LaTeX, preparada para ser introducida en un documento de este estilo, mediante una llamada a *latexString(model)*. Además, este modelo puede ser evaluado en el conjunto de test por medio de la función *evaluateTree*, que recibe el modelo y una matriz de entradas y devuelve un vector con las salidas numéricas para cada instancia. De esta forma, la salida de la función será un vector de elementos de tipo *Float64*, y se puede conseguir de la siguiente manera:

```
return evaluateTree(model, testInputs);
```

El valor de clasificación se puede conseguir aplicando un umbral en el 0, es decir, mirando el signo de cada valor, o llamando a la función *classifyOutputs* desarrollada en un ejercicio anterior.

Para el desarrollo de esta función no se permite el uso de bucles.

- Crear una función llamada *trainClassDoME*, que ejecute el algoritmo DoME para obtener modelos matemáticos a partir de un *dataset*, aplicando la estrategia “uno contra todos” en caso de que se quiera clasificar en más de dos clases. Esta función recibe como parámetros:
    - *trainingDataset*, de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}}*, con una tupla con el conjunto de entrenamiento. El segundo valor de la tupla será una matriz de valores booleanos, con tantas columnas como clases en las que haya que realizar la clasificación.
    - *testInputs*, de tipo *AbstractArray{<:Real,2}*, con las entradas del conjunto de test a las que aplicar los modelos obtenidos.
    - *maximumNodes*, de tipo *Int*, con el número máximo de nodos que puede tener la expresión generada por el algoritmo. Igual que antes, valores típicos oscilan entre 20 y 40, pero pueden ser superiores para problemas más complejos.

Esta función comenzará comprobando si la matriz de salidas deseadas de entrenamiento tiene solamente una columna. Si es así, esto indica que el problema es de clasificación binaria. En este caso, no es necesario ejecutar la estrategia "uno contra todos", sino que una única ejecución del algoritmo DoME será necesaria, y para ello, se llamará a la función anterior. Para hacer esta llamada, habrá que convertir la matriz de salidas deseadas en un vector, mediante las funciones *vec* o *reshape*. El resultado de la llamada a la función anterior se convertirá de nuevo en matriz con una columna, mediante la función *reshape*, y se devolverá esta matriz. Por lo tanto, en este caso devolverá una matriz de una columna con elementos de tipo *Float64*.

En caso de que el número de columnas sea mayor que 2, hay que aplicar una estrategia "Uno contra todos", que se basará en llamar a la función anterior una vez por columna.

- ¿Qué ocurre si el número de columnas es 2? ¿Puede existir este caso?

Para implementar esta estrategia, dar los siguientes pasos:

- Crear una matriz con tantas filas como instancias en el conjunto de test, y tantas columnas como clases haya (columnas en la matriz de salidas deseadas de entrenamiento), con elementos de tipo *Float64*.
- Crear un bucle que itere para cada clase, es decir, para cada columna de la matriz de salidas deseadas del conjunto de entrenamiento. En cada iteración, hacer:
    - Entrenar el algoritmo DoME llamando a la función anterior *trainClassDoME*, pasándole a la función las entradas de entrenamiento, y como salidas deseadas la columna correspondiente de la matriz de salidas deseadas, de tal forma que las salidas deseadas que se pasen como argumento sea un vector de valores booleanos (no una matriz bidimensional). En esta llamada se pasarán, además, las entradas de test y el número de nodos.
    - Asignar a la columna correspondientes de la matriz anterior los valores resultado de la llamada anterior a la función *trainClassDoME*.

La salida de esta función será esta matriz, que se rellena en el bucle para cada clase. Por lo tanto, la salida deberá ser una matriz de elementos de tipo *Float64*.

Para el desarrollo de esta función se permite un único bucle para iterar por las clases, en caso de tener un problema de clasificación con más de dos clases.

- Desarrollar otra función llamada *trainClassDoME*, basada en la anterior. Esta función recibirá

como parámetros:

- *trainingDataset*, de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{<:Any,2}}*, con una tupla con el conjunto de entrenamiento. La diferencia con la función anterior es que el segundo valor de la tupla será una matriz de valores de cualquier tipo, donde el número de valores distintos será el número de clases que se tenga.

- *testInputs*, de tipo *AbstractArray{<:Real,2}*, con las entradas del conjunto de test a las que aplicar los modelos obtenidos.

- *maximumNodes*, de tipo *Int*, con el número máximo de nodos que puede tener la expresión generada por el algoritmo. Valores típicos oscilan entre 20 y 40, pero pueden ser superiores para problemas más complejos.

Esta función deberá devolver como salida un vector de valores, uno por cada instancia de test (número de filas), siendo los valores de este vector del mismo tipo que los valores del vector de salidas deseadas de entrenamiento.

Esta función comenzará calculando cuántas clases se tienen mediante

```
classes = unique(trainingTargets);
```

Además, será necesario crear un vector con tantos elementos como instancias en el conjunto de test, cuyos valores sean del mismo tipo que el vector de salidas deseadas de entrenamiento, mediante

```
testOutputs = Array{eltype(trainingTargets),1}(undef, size(testInputs,1));
```

Una vez se tiene, se llamará a la función *trainClassDoME* anterior. Dado que esta necesita que las salidas deseadas se le pasen como una matriz de valores booleanos, será necesario hacer una llamada a la función *oneHotEncoding* desarrollada en un ejercicio anterior, pasándole como segundo argumento el vector *classes* previamente calculado.

```
testOutputsDoME = trainClassDoME(
      (trainingInputs, oneHotEncoding(trainingTargets, classes)),
      testInputs, maximumNodes);
```

Como resultado, se tendrá un vector o una matriz de valores de tipo *Float64*. Para asignar las clases a las que pertenece cada instancia, se utilizará la función *classifyOutputs* desarrollada en un ejercicio anterior, que toma una matriz con las certidumbres en cada clase y la

transforma en una matriz de valores booleanos con los valores de clasificación. Es importante especificar el umbral en 0, porque en el caso de que haya un vector el signo de cada componente se usará para la clasificación.

```
testOutputsBool = classifyOutputs(testOutputsDoME; threshold=0);
```

Como resultado de esta llamada, se tendrá una matriz de valores booleanos con la clasificación en las distintas clases. Aquí se distinguen dos casos:

- Si hay una o dos clases (*length(classes)<=2*), entonces la llamada a la función anterior devolverá una matriz con una columna, que podrá ser convertida a vector. El vector de salidas de test se rellenará con los valores de la primera o la segunda clase según los valores de este vector, positivos o negativos respectivamente. Hay que tener en cuenta que, si sólo hay una clase, todos los valores serán positivos, y no se podrá acceder a la segunda clase, puesto que daría error:

```
testOutputsBool = vec(testOutputsBool);
testOutputs[  testOutputsBool] .= classes[1];
if length(classes)==2
     testOutputs[.!testOutputsBool] .= classes[2];
end;
```

- Si hay más de dos clases, será necesario iterar por cada clase, haciendo algo similar: asignar en ese vector la etiqueta correspondiente en los índices de las instancias que han sido clasificadas en esa clase, de forma similar a

```
testOutputs[testOutputsBool[:,numClass]] .= classes[numClass];
```

Finalmente, se devolverá este vector de salidas en el conjunto de test.

Para el desarrollo de esta función se permite un único bucle para iterar por las clases, en caso de tener un problema de clasificación con más de dos clases.

Al final de este documento se incluyen las firmas de las funciones a realizar en este ejercicio.

Aprende Julia:

La línea de programación defensiva que permite asegurarse de que todas las etiquetas de los vectores de salidas y salidas deseadas están incluidas en el vector de clases es la siguiente:

```
@assert(all([in(label, classes) for label in vcat(targets, outputs)]));
```

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 confusionMatrix(outputs::AbstractArray{Bool,2},
       targets::AbstractArray{Bool,2}; weighted::Bool=true)
function confusionMatrix(outputs::AbstractArray{<:Real,2},
       targets::AbstractArray{Bool,2}; threshold::Real=0.5, weighted::Bool=true)
function confusionMatrix(outputs::AbstractArray{<:Any,1},
       targets::AbstractArray{<:Any,1},
       classes::AbstractArray{<:Any,1}; weighted::Bool=true)
function confusionMatrix(outputs::AbstractArray{<:Any,1},
       targets::AbstractArray{<:Any,1}; weighted::Bool=true)

function trainClassDoME(trainingDataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,1}},
       testInputs::AbstractArray{<:Real,2}, maximumNodes::Int)
function trainClassDoME(trainingDataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}},
       testInputs::AbstractArray{<:Real,2}, maximumNodes::Int)
function trainClassDoME(trainingDataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{<:Any,1}},
       testInputs::AbstractArray{<:Real,2}, maximumNodes::Int)

function printConfusionMatrix(outputs::AbstractArray{Bool,2},
       targets::AbstractArray{Bool,2}; weighted::Bool=true)
function printConfusionMatrix(outputs::AbstractArray{<:Real,2},
       targets::AbstractArray{Bool,2}; weighted::Bool=true)
printConfusionMatrix(outputs::AbstractArray{<:Any,1},
       targets::AbstractArray{<:Any,1},
       classes::AbstractArray{<:Any,1}; weighted::Bool=true)
function printConfusionMatrix(outputs::AbstractArray{<:Any,1},
       targets::AbstractArray{<:Any,1}; weighted::Bool=true)
```