

# 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 sigmoidea 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:

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

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 Soporte Vectorial (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.

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  $i$ -ésimo problema, la clase  $i$  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 cuya neurona de salida 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	$[-\infty, \infty]$	<i>softmax</i>	[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. Una asignación similar se puede construir con los patrones que pertenecen a esa clase y los que pertenecen al resto de clases. De esta forma, desde el punto de vista 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 no 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 mutuamente excluyentes.

En esta práctica, se pide:

- Desarrollar el código necesario para realizar una estrategia “uno contra todos”.
  - A pesar de que no es necesario desarrollarlo para realizar multiclase con RR.NN.AA., se utilizará en futuras prácticas.
- Los que estéis haciendo un trabajo de clasificación multiclase, aplicar este código al problema a resolver

- Dado que hasta el momento no se han visto más modelos que redes de neuronas, aplicar el código usando redes con salida binaria (una única neurona de salida) a modo de prueba. En condiciones normales de entrenamiento, se usará la capacidad de las RR.NN.AA. de realizar clasificación multiclase.
- 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 deseadas (*targets*) y salidas del modelo (*outputs*), ambas de elementos booleanos y tamaño  $N \times L$ , donde  $N$  es el número de patrones y  $L$  es el número de clases. Por tanto, es necesario especificar el tipo de los parámetros para poder sobrecargar la función anterior.
  - Para poder usar esta función directamente con las salidas de una RNA multiclase, es necesario realizar dos modificaciones a estas salidas:
    - Convertir las salidas numéricas de la RNA a valores booleanos según se explica en este documento.
    - Trasponer la matriz de salidas para que cada patrón esté situado en cada fila.
  - Añadir una línea de programación defensiva que se asegure de que en cada fila hay uno y sólo un valor positivo. Realizar esto sin hacer bucles. Para hacer esto, consultar la función *sum*, que puede recibir entradas booleanas, y el parámetro opcional *dims* que puede recibir.
  - Añadir una línea de programación defensiva que se asegure de que las matrices de entradas tienen más de dos columnas.
    - ¿Por qué no son válidas las matrices de dos columnas?
  - A la hora de medir las métricas, esto se puede hacer mediante llamadas a la función desarrollada en la práctica anterior. En este punto, se puede realizar un bucle que itere sobre el número de clases tomando las columnas correspondientes a ambas matrices y pasándoselas a la función de la práctica anterior.

- Sobrecargar esta función desarrollando otra del mismo nombre que realice la misma tarea, pero esta vez tome como entradas dos vectores (*targets* y *outputs*) de la misma longitud, cuyos elementos sean de cualquier tipo (*Any*), además del parámetro adicional que permita calcular las métricas de las formas *macro* y *weighted*. Los elementos de estos vectores representan las clases, representados de distintas formas. Por ejemplo, las clases pueden ser [*“perro”*, *“gato”*, 3].
  - Es necesario especificar también los tipos de los vectores de entrada.
    - ¿Qué tipo tendrán estos argumentos?
  - Como es obvio, es necesario que todas las clases de salida (vector *outputs*) estén incluidas en las clases de las salidas deseadas (vector *targets*). Incluir, por tanto, una línea de programación defensiva que asegure esto.
    - 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.
  - El código de esta función debería, para calcular las métricas para cada clase, realizar llamadas a la función desarrollada en la práctica anterior, del mismo nombre, que recibe vectores valores booleanos. Para ello, se puede iterar por cada clase, tomar las coincidencias de salidas y salidas deseadas (en forma de vectores booleanos) y realizar llamadas a la función de la práctica anterior.
- Sobrecargar nuevamente esta función, pero esta vez añadiendo la posibilidad de que las salidas no sean arrays de valores booleanos, sino de valores reales, y añadiendo un parámetro adicional con un valor por defecto que sea el umbral. Esta función debería aplicar el umbral y llamar a una de las anteriores.

---

### Aprende Julia:

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

```
@assert(all([in(output, unique(targets)) for output in outputs]))
```