Introducción al Aprendizaje Automático y al lenguaje Julia

Aprendizaje Automático

Los modelos de Aprendizaje Automático son sistemas que permiten resolver problemas que no son abarcables mediante técnicas de programación clásicas. Aunque en clase de teoría se estudiarán más tipos, los problemas a resolver mediante el uso de estas técnicas suelen ser fundamentalmente de clasificación o regresión:

- Problemas de clasificación: desarrollar un modelo que, a partir de ciertas características de un elemento, sea capaz de clasificarlo en una de varias clases conocidas.
- Problemas de regresión: desarrollar un modelo que, a partir de ciertas características de un elemento, sea capaz de predecir uno o varios valores numéricos del elemento.

La Fig.1 muestra esquemáticamente cómo es el sistema que se quiere obtener.

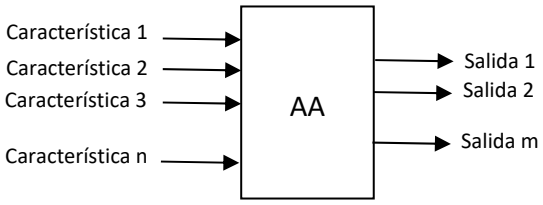


Fig. 1. Sistema a obtener

Por tanto, es necesario identificar claramente 2 cosas distintas:

- Las características de cada elemento que se utilizarán como entrada al modelo, que suelen ser valores numéricos.
- La(s) salida(s) que se quiere predecir, que suelen ser valores numéricos en el caso de problemas de regresión o booleanos en el caso de problemas de clasificación.

Estos modelos aprenden tomando un conjunto de ejemplos (patrones) en los que para cada uno de ellos ya se conoce la salida que se quiere obtener (salidas deseadas). En base al análisis de estos patrones, el modelo es capaz de determinar la relación entrada/salida existente en ese conjunto de patrones, y se puede aplicar.

Es decir, el conjunto de patrones no es más que un conjunto de instancias, y para cada instancia se

tienen los valores de entradas al modelo, y la salida que debe devolver. Por tanto, se pueden construir dos matrices distintas: una para las entradas, y otra para las salidas deseadas. Si se toman L patrones (L instancias del mundo real), y de cada instancia se toman N características (entradas), y se quiere predecir el valor de M valores (salidas deseadas), se tendrá una matriz de LxN con las entradas, y otra de LxM con las salidas deseadas, como se puede ver en la Fig. 2.

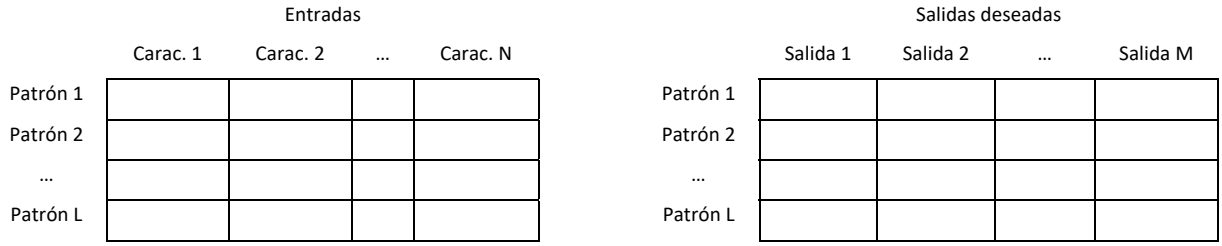


Fig. 2. Matrices con las entradas y salidas deseadas para resolver un problema.

Pasándole estas matrices a una función de entrenamiento, esta es capaz de devolver un modelo que haya aprendido la relación entrada/salida de esas instancias, con cierto nivel de error.

De esta manera, un modelo es capaz de resolver problemas complejos del mundo real que consistan en determinar relaciones entrada/salida desconocidas. Estos problemas no son resolubles mediante programación clásica, puesto que no se puede determinar un algoritmo que determine esta relación.

Para realizar esto, como ya se explica más arriba, es necesario partir de un conjunto de datos, para lo cual pueden descargarse distintas bases de datos de internet. Alguna de estas bases de datos será proveídas por los profesores para que podáis hacer distintas pruebas, concretamente la de flores iris, que es un conocido problema de clasificación.

Generalmente, los datos que se presentan en las bases de datos necesitan algún tipo de preprocesado básico. En esta práctica se verán fundamentalmente dos tipos de preprocesado básico: codificación de valores categóricos y normalización de los datos.

Con respecto al tratamiento de valores categóricos, esta es una etapa muy común, puesto que muchos modelos como las RR.NN.AA. no aceptan entradas ni salidas categóricas, sino que únicamente trabajan con valores numéricos. En cambio, muchas de las bases de datos presentan entradas y/o salidas categóricas en lugar de ser numéricas. Por lo tanto, para que los modelos que sólo aceptan entradas y salidas numéricas puedan procesarlos, es necesario convertir estos valores categóricos en numéricos, para lo cual hay tres opciones:

- Si solamente hay dos categorías, por ejemplo verdadero/falso, verde/azul, madera/metal o

caro/barato, ese atributo se transforma en otro único atributo, que toma el valor 0 para una categoría y 1 para otra.

- Si hay más de dos categorías, por ejemplo rojo/verde/azul, madera/metal/plástico o coche/barco/avión/tren, se transforma en tantos atributos como posibilidades haya, uno para a cada categoría, con valor 1 para aquellas instancias que pertenezcan a ella y 0 para las que no. Por ejemplo, en el caso rojo/verde/azul los patrones con valor "rojo" pasarán a ser (1, 0, 0), los "verde" (0, 1, 0) y los "azul" (0, 0, 1).

- Existe una tercera posibilidad, cuando hay más de dos categorías, que consiste en convertirlas en un único número real. Por ejemplo, A/B/C/D podría convertirse en 0/0.33/0.66/1. Sin embargo, este caso solamente es interesante cuando en el mundo real exista un orden A < B < C < D.

Con respecto a la normalización de los datos, el entrenamiento de un modelo será mucho más rápido si las entradas proporcionadas están en la misma escala, es decir, si se le ahorra al modelo el tener que aprender la relación entre las escalas en las que se mueve cada atributo. A este proceso de convertir las entradas para que todas estén en el mismo intervalo se denomina **normalización**, y constituye uno de los tipos de preprocesado más común e importante. Mediante este tipo de preprocesado se consigue que un modelo más sencillo pueda resolver problemas más complejos que sin él, puesto que no necesita emplear parte de la misma para aprender la relación entre las escalas de los atributos de entrada.

- ¿Sería necesario realizar este preprocesado cuando las entradas son los valores de intensidad de cada pixel en una imagen en blanco y negro? ¿Por qué?

Existen muchos otros tipos de preprocesado, como puede ser limpieza de ruido, análisis PCA, etc. Algunos de ellos se verán en una asignatura posterior. Con respecto a la normalización, en clases de teoría se explican más, pero en la práctica se suele usar uno de estos dos tipos:

- Normalización entre máximo y mínimo. Para cada atributo, se toma el menor (*min*) y el mayor (*max*) valor que se tiene, y se cambian todos los valores *v* para pasarlos al nuevo intervalo *[nuevomin, nuevomax]* de la siguiente manera:

$$v' = \frac{v - \min}{\max - \min}(\text{nuevomax} - \text{nuevomin}) + \text{nuevomin}$$

  Generalmente se suele pasar a un intervalo entre [0, 1], por lo que la ecuación queda más

sencilla:

$$v' = \frac{v - \min}{\max - \min}$$

Este tipo de normalización es adecuado cuando se tiene la certeza de que los datos están acotados (tanto superior como inferiormente), es decir, están dentro de un intervalo. Lo que está haciendo, en la práctica, es cambiar el intervalo de los datos por el intervalo [0, 1] y hacer una correspondencia para cada dato a su nuevo valor dentro del nuevo intervalo. Sin embargo, si se sospecha que uno de estos datos podría salirse del intervalo y tomar un valor excesivamente alto o bajo, esta transformación puede ser muy nociva. En este caso, a este valor *atípico* se le asignaría como nuevo valor el 1 si es excesivamente alto, y el resto de valores oscilarían cercanos al 0 con poca diferencia entre ellos, o el 0 si es excesivamente bajo, y el resto de valores oscilarían cercanos al 1 con poca diferencia entre ellos. Si se sospecha que puede haber casos como este, se suele optar por la siguiente forma de normalización.

  - En este tipo de normalización, si *min*=*max*, se puede realizar otro preprocesado distinto en este atributo, ¿en qué consistiría?

- Normalización de media 0. Como se ha dicho, esta forma de normalización es más robusta ante valores atípicos. Para cada atributo, se toma la media y desviación típica de todos los valores que toma, y se hace una transformación sencilla:

$$v' = \frac{v - \mu}{\sigma}$$

  De esta manera, los valores de cada atributo pasarán a tener una media 0 y una desviación típica de 1. Algunos valores saldrán de este intervalo, pero esto no supone ningún problema para la RNA, que sencillamente acepta como entradas valores reales.

Es importante tener en cuenta que, se aplique la forma de normalizar que se aplique, esta se realice de forma independiente para cada atributo. Es decir, si se tiene una base de datos con N patrones y L atributos, habría que realizar L normalizaciones distintas.

- Sabiendo que los modelos de Aprendizaje Automático en general suponen que los patrones se distribuyen en filas, pero en el mundo de las RR.NN.AA. se distribuyen en columnas, ¿en qué casos habría que normalizar cada fila por separado y en qué casos habría que normalizar cada columna por separado?

En general, el tener el conocimiento sobre la naturaleza de cada atributo suele ser de gran utilidad para conseguir modelos que ofrezcan mejores resultados. Cuanta más información sobre los datos se "introduzca" dentro del modelo, mejor funcionará este. Un buen ejemplo de introducir información sobre los datos es la normalización de los mismos. Esto podría llevarse al extremo y escoger una forma de normalización distinta, la que se crea que es más adecuada, para cada uno de los atributos. Esto puede llevar a tomar la decisión, por ejemplo, de normalizar el atributo "temperatura" entre máximo y mínimo, y el atributo "distancia" mediante media 0. De todas formas, en la mayoría de los casos se suele optar por una de las dos formas de normalización y aplicarla a todos los atributos de entrada de la RNA.

También es importante tener en cuenta que este proceso ocurre en las salidas de la RNA. Es decir, si las salidas están en intervalos distintos, la RNA tiene que aprender esto también, con lo que se puede "ayudar" a la RNA mediante la normalización de los datos de salida. Este es un proceso que se realiza en problemas de regresión, pero no en problemas de clasificación.

- ¿Por qué no se realiza en las salidas de problemas de clasificación?

Importante: De esta forma, las entradas (y salidas deseadas) que se aplican a un modelo ya no son los datos originales, sino que han sido transformados. Por lo tanto, una vez entrenado, un modelo no está preparado para que se le pasen los datos originales, sino que si se desea aplicar los datos, estos tendrán que ser transformados de la misma manera. Por este motivo, se aplique la forma de normalizar que se aplique, es necesario guardar los parámetros usados en la normalización **para cada atributo** (máximo y mínimo o media y desviación típica). De igual manera, si se ha normalizado la salida deseada (problemas de regresión), el modelo habrá aprendido a emitir una salida normalizada, por lo que habrá que desnormalizarla, lo cual conlleva nuevamente que habrá que guardar los parámetros de normalización de las salidas deseadas. En resumen, para aplicarle nuevos datos a un modelo, el proceso será el siguiente:

1. Normalizar los datos según los parámetros de normalización que ya se tienen y que se utilizaron en el conjunto de entrenamiento.

2. Aplicar los datos normalizados al modelo.

3. En caso de problemas de regresión, desnormalizar las salidas del modelo.

En esta primera práctica se desarrollará el código necesario para poder aplicar distintas técnicas de Aprendizaje Automático. Para ello, durante varias semanas se plantearán ejercicios que se integrarán en un código que se irá realizando por tanto de forma incremental durante estas semanas. Este

código se podrá probar con una de las bases de datos dadas por los profesores, o bien por cualquier otra descargada por el equipo de trabajo en internet. Al final de estos ejercicios, se entregará un archivo de Julia con todas las funciones desarrolladas.

En este primer ejercicio, se pide:

- Comprender el problema que se quiere resolver.
- Conocer la base de datos de la que se parte.
    - ¿Cuántos patrones se tiene?
    - ¿Cuántos atributos tiene cada patrón? ¿Son todos relevantes? ¿Qué describe cada atributo?
    - ¿Hay entradas o salidas categóricas? ¿Cómo se van a procesar?
- Tener un primer contacto con Julia, instalar los paquetes necesarios y aprender los conceptos más básicos de Julia.
    - Para esto, al final de este enunciado se adjunta un tutorial de Julia.
- Cargar la base de datos en Julia.
    - ¿Qué dimensiones tienen las matrices?
    - Para cargar los datos, puede ser útil la función *readdata* del paquete XLSX o la función *readdlm* del paquete DelimitedFiles, según cómo sea el archivo con los datos.
- Desarrollar el código que permita codificar los valores categóricos que tenga en valores booleanos, distinguiendo los dos casos (tener una o dos categorías, y tener más de dos categorías).
    - El caso a contemplar de estos dos posibles debe ser automáticamente detectado en el código.
    - Si se tiene una única categoría, esto se trata igual que si hubiese dos categorías, devolviendo un vector de valores booleanos con todos igual a *true*.
    - Este código debe partir de un vector (matriz unidimensional) con los valores de un atributo o salida deseada, y devolver un vector (matriz unidimensional) o matriz

(bidimensional) de salida, en función de la forma de codificación de ese atributo o salida deseada.

  - Desarrollar este código sin utilizar bucles que recorran todos los patrones. A pesar de que es posible hacer todo este código sin bucles, se permite hacer un bucle que recorra las clases o los atributos, pero sólo se puede realizar este bucle.

  - Para hacer este código, puede ser útil la función *unique*.

  - Este código será aplicado a cada una de las entradas/salidas categóricas que tenga el problema escogido. Para ello, en la siguiente práctica este código será convertido a una función.

- Desarrollar el código que, a partir del conjunto de datos de entrada, extraiga los valores de máximo, mínimo, media y desviación típica de cada columna. Para ello, consultar las funciones *minimum*, *maximum*, *mean* y *std*. Estas dos últimas necesitan que previamente se haya cargado el paquete *Statistics* (mediante *using Statistics*). Además, estas funciones aceptan el *keyword* adicional *dims*, con el que se especifica al final de este tutorial. Si se utiliza *dims* de forma apropiada, el resultado de las llamadas a estas funciones serán matrices de una fila y tantas columnas como atributos, que contendrán estos valores de mínimo, máximo, media y desviación típica.

  Una vez extraídos estas matrices de una fila, utilizar una de las dos formas explicadas aquí para normalizar las entradas de la base de datos. Esta tarea se puede realizar de forma muy sencilla haciendo *broadcast* sencillos de operaciones de restas y divisiones, sin necesidad de realizar bucles, como se muestra en el tutorial de Julia de esta práctica.

  Además, será necesario contemplar los siguientes casos:

  - Si se normaliza entre máximo y mínimo y en algún atributo el mínimo es igual al máximo.

  - Si se normaliza mediante media y desviación típica y en algún atributo la desviación típica es 0.

  En cualquiera de estos dos casos se está dando la misma situación: todos los patrones toman el mismo valor para un atributo. En este caso, una solución común es eliminar el atributo, puesto que no aporta información. Otra posibilidad, más sencilla, es asignarle un valor constante, por ejemplo, el valor de 0.

Una función que puede ser útil para convertir una matriz bidimensional con una fila fila o una matriz bidimensional con una columna en un vector es la función *vec*. Con ella, se puede referenciar fácilmente los elementos de la matriz

- Como resultado de esta práctica, después de aplicar este código a las entradas y/o salidas categóricas, se debería disponer de dos matrices (entradas y salidas deseadas).

Aprende Julia:

Para la implementación de las soluciones, existen distintas posibilidades en cuanto al lenguaje de programación. Entre la gran cantidad de opciones disponibles, se han tomado en consideración tres posibilidades:

- Matlab. Este es un lenguaje científico con muchos años de trayectoria, y por lo tanto tiene como ventaja que posee una gran cantidad de módulos (llamados *Toolboxes*) para casi cualquier operación que se desee realizar, junto con una excelente documentación. Esto le convierte en un lenguaje muy indicado para iniciarse y aprender. Sin embargo, tiene como principal inconveniente que es necesario adquirir una licencia para utilizarlo. Si bien la Facultad de Informática y la Universidade da Coruña posee licencia para su uso, el hecho de que sea necesario adquirirla para utilizarlo de forma profesional ha hecho que muchas empresas no se decanten por esta opción, por lo que en la práctica a nivel empresarial no es tan utilizado como python.

- Python. Sin duda, es el más utilizado. Es un lenguaje moderno, y sencillo, con una gran cantidad de módulos y abundante documentación, aunque sin llegar al número ni calidad de Matlab. Es gratuito y de código abierto, y de propósito general, lo que ha hecho que sea uno de los lenguajes más utilizados en la actualidad en el mundo empresarial. Además, una de las primeras librerías de *Deep Learning*, llamada *Tensorflow*, fue desarrollada por Google para este lenguaje, lo que ha aumentado drásticamente la comunidad de desarrolladores de aplicaciones de aprendizaje máquina en este lenguaje. Librerías como Scikit-Learn permiten también el uso de otras técnicas de aprendizaje máquina como árboles de decisión o Máquinas de Soporte Vectorial. El mayor problema que tiene este lenguaje es que no es un lenguaje científico sino de propósito general, y la programación vectorial no está soportada de forma nativa, sino a través de la librería numpy, lo que conlleva una considerable pérdida de rendimiento.

- Julia. Este es un lenguaje emergente, de vida muy corta, y totalmente científico, desarrollado en el Instituto Tecnológico de Massachussets. Su primera versión estable tiene unos pocos años de vida, y actualmente se encuentra bajo un intenso desarrollo. Por este motivo, no posee el mismo número de módulos que Matlab o Python, aunque la cantidad que posee está aumentando muy rápidamente, puesto que el lenguaje es gratuito y de código abierto. Este lenguaje ha sido desarrollado como un punto intermedio entre Matlab, como lenguaje científico, y python, como lenguaje sencillo y de código abierto, con una velocidad de ejecución superior a ambos. Como principales inconvenientes tiene que la comunidad de desarrolladores no es tan grande como la de python, y el número de módulos no es tan grande como el de Matlab, y, dado su corto período de tiempo, todavía no ha hecho presencia en el mundo empresarial.

Entre estos tres lenguajes se ha escogido Julia para realizar las prácticas de esta asignatura, puesto que el tiempo de cómputo de los algoritmos de aprendizaje máquina es un factor clave en las prácticas, y Julia es el que ofrece mejor rendimiento.

Como se ha indicado, a nivel empresarial es un lenguaje que todavía no tiene presencia, sin embargo, esto se ve paliado por dos factores importantes:

- Al haber adquirido destreza en programar en lenguaje python en otras asignaturas, esta asignatura supone la oportunidad de aprender un lenguaje científico que complete los conocimientos de programación.

- A pesar de que a nivel empresarial python sea el lenguaje más utilizado, la librería Scikit-Learn está disponible también en Julia, así que su aprendizaje en esta asignatura implicaría que podrían utilizarla también en su trabajo en una empresa en el lenguaje python sin esfuerzo, al tener conocimientos tanto de dicha librería como de dicho lenguaje.

Como ya se ha indicado, Julia es un lenguaje gratuito y de código abierto, y puede descargarse de https://julialang.org/downloads/. Únicamente con esta descarga ya se puede utilizar Julia. Sin embargo, si se quiere tener un IDE para Julia, una posibilidad es utilizar una extensión para Visual Studio Code. Para ello, primero es necesario tener instalado VS Code (disponible en https://code.visualstudio.com/download), y posteriormente instalar la extensión para Julia. Las instrucciones para instalar y configurar esta extensión para Visual Studio Code están disponibles en https://code.visualstudio.com/docs/languages/julia. Una vez configurada, para arrancar un REPL de Julia se pueden abrir los comandos mediante Ctrl+Shift+P, y en el mismo ejecutar “Julia: Start REPL”. De esta manera, se puede usar Julia de la misma manera que entrando en la versión instalada de

Julia, pero con las ventajas de contar con un entorno de desarrollo integrado.

En esta asignatura se utilizarán distintos paquetes de Julia. Algunos de ellos ya vienen instalados, por ejemplo el paquete Statistics. Para utilizarlo simplemente se puede escribir al inicio de un script "using Statistics". Si solamente se va a usar una o varias funciones y no se desea cargar todo el paquete, se puede indicar esto, escribiendo algo como "using Statistics: mean". Otros paquetes no vienen por defecto en la distribución de Julia, y es necesario instalarlos. Algunos de estos paquetes son los siguientes:

- XLSX: Permite cargar archivos de Excel en Julia.
- Flux: Permite entrenar Redes de Neuronas Artificiales en Julia.
- ScikitLearn: Librería de python que provee de un interfaz uniforme para entrenar y utilizar modelos de Aprendizaje Automático, así como otras utilidades. En Julia, estas funcionalidades están disponibles en la misma librería.
- SymDoME: Permite ejecutar el algoritmo DoME en Julia, para resolver problemas de Regresión Simbólica y Clasificación.
- Plots: Permite representar gráficas en Julia.
- Images: Permite trabajar con imágenes.
- GeneticProgramming: Permite desarrollar Algoritmos Genéticos en Julia.
- FileIO: Permite operar con archivos.
- JLD2: Permite cargar y guardar variables de Julia en archivo. Es necesario tener instalado el paquete FileIO.
- MAT: Permite cargar variables de Matlab en Julia.

La primera vez que se usan, estos se precompilan. Por ejemplo, si los datos a cargar están en formato de hoja Excel, para cargarlos se puede usar la función *readdata* del paquete XLSX. Para lo cual, este paquete debe estar previamente instalado, lo que se puede hacer escribiendo en la línea de comandos lo siguiente:

```
import Pkg; Pkg.add("XLSX");
```

Y para usar la función en un script, se puede escribir "*using XLSX: readdata*" al principio del script.

Para realizar esta práctica, es necesario realizar dos llamadas para cargar las dos matrices. Si estos datos están en una hoja Excel, estas llamadas serán algo similar a esto:

```
inputs = readdata("iris.xlsx","inputs","A1:D150");

targets = readdata("iris.xlsx","targets","A1:C150");
```

De esta forma, se cargan dos variables en memoria, una con la matriz de entradas y otra con la matriz de salidas deseadas.

En otras ocasiones, los datos estarán en un archivo de texto con algún tipo de delimitador. Esto ocurre, por ejemplo, con el archivo "iris.data" que se puede descargar de https://archive.ics.uci.edu/dataset/53/iris Para ello es útil el paquete DelimitedFiles. Este paquete se puede instalar de la forma habitual con

```
import Pkg; Pkg.add("DelimitedFiles");
```

Igualmente, para utilizarlo solamente hay que escribir al principio del *script* lo siguiente:

```
using DelimitedFiles
```

Una vez cargado, para cargar una base de datos, esto se puede hacer con la siguiente línea:

```
dataset = readdlm("iris.data",',');
```

Como se puede ver, el primer parámetro es el nombre del archivo, mientras que el segundo es el delimitador o delimitadores que se van a utilizar. En este caso, se carga toda la base de datos en una única variable llamada *dataset*, que será una matriz bidimensional. En la mayoría de las ocasiones será necesario separar las entradas de las salidas deseadas. Esto se puede hacer con las siguientes líneas:

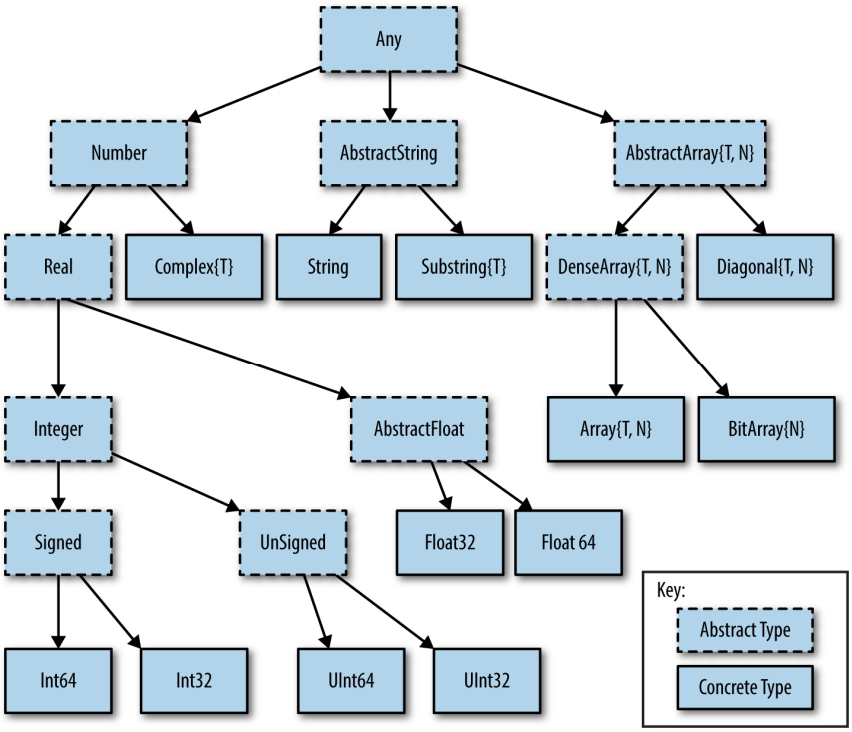
```
inputs = dataset[:,1:4];

targets = dataset[:,5];
```

El operador de corchetes permite referenciar un dato o una parte de una matriz, el primer elemento se refiere a las filas y el segundo a las columnas. Cuando se pone dos puntos (:), se dice que se quiere referenciar todas las filas o todas las columnas, devolviendo por lo tanto una matriz en lugar de un elemento. Esto será descrito más en profundidad más adelante en este tutorial. En el caso particular de la base de datos de flores iris, las cuatro primeras columnas son las entradas y la quinta se

corresponde con la salida deseada, por ese motivo en estas dos líneas se pueden ver los valores 1:4 y 5. En otra base de datos, las columnas serían distintas.

Es importante tener en cuenta que todas las variables tienen un tipo. Julia provee de una jerarquía de tipos donde el tipo de la raíz se denomina *Any*. Es decir, cualquier variable será de tipo Any. En este caso, al leer de una hoja de cálculo una gran cantidad de datos que pueden ser de naturaleza distinta (números, fechas, cadenas de caracteres, etc.), devuelve una matriz bidimensional donde cada elemento es de tipo *Any*. En Julia, esto se representa como *Array{Any,2}*, donde *Any* indica el tipo de cada elemento de la matriz, y el 2 indica el número de dimensiones. En la siguiente imagen se puede ver la jerarquía de varios tipos comunes en Julia (Fuente: *Learning Julia: Abstract, Concrete, and Parametric Types by Spencer Russell, Leah Hanson*):



Como se puede ver, por ejemplo, el tipo *Int64* es subtipo de *Signed*, *Integer*, *Real*, *Number* y *Any*. Para ver de qué tipo es una variable, se puede usar la función *typeof*; para comprobar si una variable es de un tipo determinado, se puede usar la función *isa*. Los tipos, a su vez, también tienen un tipo determinado, que es *DataType*. El tipo de *DataType* es *DataType*, que es subtipo de *Any*. Por lo tanto, cualquier elemento, incluidos los tipos, son de tipo *Any*, puesto que todos los tipos son subtipos de *Any*.

- ¿Cuál será el resultado de las siguientes llamadas?

  - `typeof(Array{Float64,2})`

- `typeof(DataType)`
- `typeof(Any)`
- `isa(DataType, Any)`
- `isa(Any, Any)`
- `isa(Array{Float32,2}, Any)`
- `isa(typeof(Array{Float64,2}), Any)`

Generalmente los tipos más utilizados en Aprendizaje Automático para almacenar los datos numéricos serán *Float32* o *Float64*. *Float32* es muy utilizado en el mundo del Aprendizaje Automático porque es el tipo de datos que utiliza la mayoría de las tarjetas gráficas, y es el tipo que usaremos en esta asignatura, por proveer de una precisión suficiente para los trabajos que se van a realizar.

Por lo tanto, es necesario convertir los datos que usaremos, de *Array{Any,2}* a *Array{Float32,2}*. Una posibilidad es usar la función *convert*, que intenta hacer una conversión a un tipo especificado, con lo que podría hacerse algo como:

```
inputs = convert(Array{Float32,2},inputs);

targets = convert(Array{Float32,1},targets);
```

En este último ejemplo se han convertido las salidas deseadas a una matriz bidimensional de valores reales, lo cual es útil en problemas de regresión con varias salidas. En el problema de flores iris con el que se está trabajando, esta línea daría error puesto que targets es de tipo *Array{Any,1}*, donde cada elemento es un *String*, que no puede ser convertido a *Float32*. Tened en cuenta que esto habría que adaptarlo al problema en cuestión. Por ejemplo, si es un problema de clasificación en dos clases y el archivo tiene valores numéricos de 0 y 1 para cada clase, esto se podría convertir en valores booleanos con algo como:

```
targets = convert(Array{Bool,1}, targets);
```

Si el problema es de clasificación y hay más de dos clases, habría que realizar una conversión más compleja, en una matriz donde el número de columnas es igual al número de clases y para cada patrón se tiene un valor de *true* en la columna correspondiente a la clase a la que pertenece y *false* en el resto, como se plantea en los ejercicios de esta práctica.

Otra posibilidad es forzar un tipado usando el propio tipo como si fuese una función. Por ejemplo, se puede especificar que un número sea de un tipo determinado escribiendo algo como *Float64(8)*, *Float32(8)*, *Int64(8)*, *UInt32(8)*, etc. Sin embargo, no se puede hacer *Float64(inputs)*, porque *inputs* no es de tipo *Number*, por lo que esa conversión no es posible. Lo que se quiere no es forzar el tipo de la matriz, sino crear una nueva matriz donde cada elemento sea el resultado de forzar el tipo del elemento correspondiente de la matriz inicial, es decir, hacer un ***broadcast*** del forzado de tipo.

Uno de los puntos en los que Julia es más potente es en el manejo de matrices multidimensionales. Julia permite, entre otras cosas, aplicar funciones a todos los elementos de una matriz y construir la matriz resultado de una forma automatizada. En este caso, se dice que se ha hecho un *broadcast* de esa función en toda la matriz. A continuación, un ejemplo sencillo: se define la función que calcula el cuadrado de un número de la siguiente manera:

```
cuadrado(x::Real) = x*x
```

En este caso, se ha especificado que el tipo del argumento es *Real*, con lo que se obliga a que el argumento sea *Int32*, *Int64*, *Float32*, *Float64*, etc. (si no se especifica el tipo del argumento, por defecto Julia entiende que es de tipo *Any*). Se está, por tanto, definiendo una función **entre números, no entre matrices**. Por ejemplo, la siguiente llamada daría un error:

```
cuadrado([1 2 3])
```

puesto que esta función está definida entre números y se está pasando como argumento una matriz, concretamente de una fila y 3 columnas. Sin embargo, si se desea construir una nueva matriz del mismo tamaño que la matriz original donde cada elemento sea el resultante de aplicar esta función al elemento correspondiente de la matriz original, esto se puede hacer escribiendo un punto '.' después del nombre de la función, de la siguiente manera:

```
cuadrado.([1 2 3])
```

De esta forma, se indica a Julia que se debe aplicar esa función elemento a elemento. Realizar estas operaciones de *broadcast* permite desarrollar código mucho más limpio, puesto que se evita escribir bucles, y eficiente, puesto que Julia es capaz de distribuir estas operaciones paralelas en distintos núcleos. Otra forma alternativa de realizar este proceso sería el siguiente:

```
[cuadrado(x) for x in [1 2 3]]
```

Otro ejemplo es el siguiente, con la operación suma:

```
suma(a::Real, b::Real) = a+b
```

Esta llamada dará error, puesto que no está definida la función 'suma' entre matrices:

```
suma([1 2 3],[2 3 4])
```

Sin embargo, se puede hacer *broadcast* de esta operación entre los elementos de ambas matrices, de la siguiente manera:

```
suma.([1 2 3],[2 3 4])
```

En este caso, ambas matrices tienen que tener el mismo tamaño. Incluso es posible que uno de los argumentos sea un número y el otro una matriz.

- ¿Cuál será el resultado de hacer *suma.(1,[2 3 4])* y *suma.([1 2 3],3)*?

En general, las operaciones matemáticas más comunes tienen implementados operadores de *broadcast*. Por ejemplo, si A y B son dos matrices del mismo tamaño, para operar elemento a elemento se puede hacer A.+B, A.*B, A./B, o A.-B. Otro ejemplo es hacer A.^2, donde se construye una matriz nueva donde cada elemento es el correspondiente, elevado al cuadrado.

Volviendo a los tipos, por lo tanto es equivalente escribir estas líneas:

```
inputs = Float32.(inputs);
```

```
inputs = [Float32(x) for x in inputs];
```

```
inputs = convert(Array{Float32,2},inputs);
```

Una cuestión que será de importancia a la hora de definir funciones es comprender bien el sistema de tipos de Julia. Como ya se ha dicho, cualquier elemento, incluidos los tipos, son de tipo *Any*, puesto que todos los tipos son subtipos de *Any*. Sin embargo, es necesario tener cuidado con los tipos que se basan en otros, por ejemplo, los *arrays*, en el que los elementos son de un tipo determinado. En este caso, por ejemplo, una variable que sea de tipo *Array{Float32,2}* será también de tipo *Any*, pero no de tipo *Array{AbstractFloat,2}*, *Array{Real,2}* ni *Array{Number,2}* ni *Array{Any,2}*, porque, a pesar de que *Float32* es un subtipo de *AbstractFloat*, *Real*, *Number* y *Any*, el tipo *Array{Float32,2}* no es subtipo de *Array{AbstractFloat,2}*, *Array{Real,2}*, *Array{Number,2}* o *Array{Any,2}*. Para indicar si en un tipo de este estilo sus elementos son subtipos de otros, se utilizará el operador <: como se puede ver en el siguiente ejemplo. Esto será de utilidad en las definiciones de funciones.

- ¿Cuál será el resultado de las siguientes llamadas?
    - `typeof(inputs)`
    - `isa(inputs, Any)`
    - `isa(inputs, Array)`
    - `isa(inputs, Array{Float32,2})`
    - `isa(inputs, Array{Real,2})`
    - `isa(inputs, Array{Number,2})`
    - `isa(inputs, Array{Any,2})`
    - `isa(inputs, Array{<:Real,2})`
    - `isa(inputs, Array{<:Number,2})`
    - `isa(inputs, Array{<:Any,2})`

El operador <: también se puede utilizar para comprobar si un tipo es subtipo de otro, como se puede ver en el siguiente ejemplo:

- ¿Cuál será el resultado de las siguientes llamadas?
    - `Array{Float32,2} <: Any`
    - `Array{Float32,2} <: Array`
    - `Array{Float32,2} <: Array{Any,2}`
    - `Array{Float32,2} <: Array{<:Real,2}`
    - `Array{Float32,2} <: Array{<:Any,2}`

De nuevo, esta conversión de las salidas deseadas es útil cuando el problema es de regresión. Si el problema es de clasificación con 2 clases, y nos dan estas salidas deseadas como valores numéricos 0 o 1, serían equivalentes las siguientes líneas:

```
targets = Bool.(targets);
```

```
targets = [Bool(x) for x in targets];
```

```
targets = convert(Array{Bool,1},targets);
```

Con respecto a los vectores y matrices de valores booleanos, existen dos tipos que en la mayoría de los casos se pueden usar indistintamente, que son *Array{Bool,N}* y *BitArray{N}*, donde N indica la dimensionalidad de la matriz. El tipo *Array{Bool,N}* almacena cada valor booleano como un valor de tipo *Bool*, que se representa internamente como un valor de tipo *UInt8*. Por tanto, si el *array* tiene n elementos, necesitará n bytes para almacenarlo. Por su parte, el tipo *BitArray{N}* almacena cada valor booleano como un bit, por lo que n elementos necesitan n/8 bytes para ser almacenados, una cantidad muy inferior al tipo *Array{Bool,N}*. Dependiendo de la situación, puede ser más eficiente almacenar de una forma o de la otra. En todo caso, las operaciones más comunes están definidas para ambos tipos, por lo que en la inmensa mayoría de las ocasiones son intercambiables y por lo tanto se puede usar uno u otro. De cara a la definición de funciones, es importante tener en cuenta que ambos son subtipos de AbstractArray{Bool,N}, como se puede ver en los siguientes ejemplos:

```
BitArray{2} <: AbstractArray{Bool,2}
```

```
Array{Bool,2} <: AbstractArray{Bool,2}
```

De esta manera, se tendrá una matriz con las entradas (llamada *inputs*) de tipo *Array{Float32,2}*. La matriz con las salidas deseadas (llamada *targets*) será necesario construirla dependiendo de la naturaleza del problema a resolver. A pesar de que *Array{Float32,2}* es el tipo que más se va a usar en esta asignatura, tened en cuenta que los tipos de datos en Julia son muy flexibles; por ejemplo se podría tener una variable que contenga una matriz tridimensional donde cada elemento sea un vector, el tipo sería *Array{Array{Float32,1},3}*.

- ¿Qué tipo tendrán los objetos [[]], [[8]] y [[8.]]?

De forma genérica, en Aprendizaje Automático se entiende que las matrices contienen cada instancia en cada fila, mientras que en la matriz de entradas las columnas representan los atributos, y en la matriz de salidas deseadas cada columna representa una salida distinta. Por lo tanto, ambas matrices (*inputs* y *targets*) deben tener el mismo número de filas.

Para saber el número de filas y/o columnas de una matriz se puede usar la función *size*. Esta función devuelve una tupla, con el número de elementos igual al número de dimensiones, en la que cada elemento indica el tamaño de esa dimensión. Por ejemplo, la llamada

```
size(inputs)
```

devuelve (150, 4), es decir, 150 filas y 4 columnas. También es posible llamar a esta función indicando de qué dimensión se quiere leer el tamaño. Por ejemplo,

```
size(inputs,1)
```

devolverá un valor de 150.

En este caso, de haber hecho bien la base de datos en el archivo Excel, ambas matrices deberían tener el mismo número de filas. Sin embargo, este tipo de cuestiones en muchas ocasiones deberían ser comprobados con el objetivo de encontrar posibles errores. Para eso, en muchas partes del código suele ser interesante introducir comprobaciones para verificar que todo está correcto. Cuando se ejecute esa comprobación, si no es cierta, el sistema debería dar un error. A esto se llama hacer *programación defensiva*. En el caso de Julia, esto se puede hacer con la macro *@assert*, a la que se indica la comprobación a realizar, y, de forma opcional, el mensaje de error que debería aparecer, por ejemplo:

```
@assert (size(inputs,1)==size(targets,1)) "Las matrices de entradas y salidas deseadas no tienen el mismo número de filas"
```

Como la variable targets es un vector, es decir, un *Array* de una dimensión, la anterior llamada a *size(targets,1)* podría ser sustituida por *length(targets)*.

Llegados a este punto, y una vez realizados los ejercicios de esta práctica, deberían estar cargadas dos matrices en memoria, ambas con el mismo número de filas. Importante: Al contrario que en el resto de modelos, en el mundo de las Redes de Neuronas Artificiales, se suele entender que cada instancia está en cada columna, siendo las filas de la matriz de entradas los atributos, y las filas de la matriz de salidas deseadas las salidas de la RNA.

Por lo tanto, la primera práctica junto con la introducción a Julia concluiría en este punto. Sin embargo, como ya se ha dicho, el uso de matrices junto con el *broadcast* de funciones es uno de los puntos fuertes de Julia, en los que es especialmente eficiente. Por este motivo, resulta interesante exponer aquí cómo operar con matrices, que se realiza de una forma muy similar a Matlab.

Para crear un vector se pueden escribir sus elementos entre corchetes, separados por comas, por ejemplo:

```
M = [1, 2, 3]
```

Para crear una matriz basta con escribir sus elementos entre corchetes, separando las filas por

puntos y comas (;), por ejemplo:

```
M = [1 2 3; 4 5 6]
```

Para acceder a un elemento de la matriz se indica el nombre de la matriz seguido de la fila y la columna a la que se desea acceder entre corchetes:

```
M[2,3]
```

En este caso, la matriz M declarada es de dos dimensiones, por lo que entre corchetes se han indicado dos valores. En el caso de tener una dimensionalidad distinta, habría que indicar un valor por cada dimensión. Por ejemplo, si fuese de 3 dimensiones habría que escribir M[2,3,1].

- ¿Cuál será el resultado de las siguientes llamadas? Al representar un vector como si fuera una matriz, ¿esta será una matriz fila o una matriz columna? ¿La tercera llamada devuelve un vector o una matriz bidimensional?
  - `typeof([1,2,3])`
  - `typeof([1;2;3])`
  - `typeof([1 2 3])`
  - `typeof([1 2 3; 4 5 6])`

El operador dos puntos (:) es el operador de rango, que vale para crear vectores. En Matlab J:K es equivalente a crear el vector [J, J+1, ..., K] siempre que J < K. Además, J:D:K es equivalente a [J,J+D, J+2* D,..., K]. Por ejemplo, en Matlab son equivalentes las siguientes operaciones:

```
1:3
[1 2 3]
```

Sin embargo, en Julia existe un pequeño cambio en este aspecto, si bien la operabilidad sigue siendo la misma. El cambio en Julia consiste en que J:D:K no crea un vector sino un elemento de tipo *UnitRange* o *StepRange*, que almacena los índices inicial y final y el incremento, y que se puede utilizar de la misma manera que un vector. De esta forma, se elimina la necesidad de crear vectores en muchas ocasiones en las que no son necesarios, por ejemplo en bucles. Además, esto se realiza de forma transparente al desarrollador, puesto que, como se ha comentado, la operabilidad es la misma que en el caso de crear vectores explícitos.

El operador dos puntos también se puede utilizar para seleccionar filas, columnas o partes de una matriz. Estos operadores se pueden utilizar junto con la palabra *end*, que indica que el rango terminará en el valor último de la fila o columna. Por ejemplo, se puede probar los siguientes ejemplos, que permiten tomar de la matriz M todas las filas y la primera columna, la primera fila y las columnas a partir de la segunda y la segunda fila y todas las columnas respectivamente:

```
M[:, 1]
M[1, 2:end]
M[2, :]
```

Al referenciar elementos de una matriz, hay que tener en cuenta que, por cada dimensión en el que no se indique un rango sino un único valor (por ejemplo, una fila o una columna), se perderá esa dimensión.

- ¿Qué dimensiones y tamaños tendrán las matrices resultado de estas tres operaciones? ¿Qué dimensión se "pierde" en cada una?

Como se indicaba, la operabilidad con este objeto es la misma que con vectores. Por ejemplo, estas dos líneas ofrecen el mismo resultado:

```
M[1, end:-1:1]
M[1, [3, 2, 1]]
```

En estos dos ejemplos, como se puede ver, se "pierde" una dimensión puesto que se está referenciando una única fila, la primera, y por lo tanto el resultado será un vector (es decir, un array de dimensión 1). Algo similar ocurre cuando a un vector (array de dimensión 1) se referencia un elemento haciendo, por ejemplo, *M[3]*, en este caso se "pierde" la dimensión que tenía y devuelve un objeto de 0 dimensiones (un valor escalar). Si en lugar de referenciar mediante un escalar se referencia mediante un rango o un vector, entonces no se "perderá" esa dimensión, sino que posiblemente disminuirá. Por ejemplo, las siguientes operaciones sobre el vector definido como *M=[1,2,3]* devuelven otro vector, aunque en el penúltimo ejemplo el vector que se devuelve tiene una mayor longitud que el original:

```
M[:]
M[1:end]
M[end:-1:1]
```

```
M[1:3]

M[[1,2,3]]

M[[1, 2, 3, 1, 2, 3]]

M[[2]]
```

En todos estos casos se devuelve un vector. Sin embargo, el último ejemplo es interesante para terminar de comprender cómo funciona el operador para referenciar partes de arrays. En este ejemplo, se devuelve un vector de un solo elemento. Se devuelve un vector porque se ha utilizado un vector para referenciar, y tiene un solo elemento porque este vector usado para referenciar tiene un solo elemento. Por lo tanto, lo que devuelve es un objeto distinto que si se hubiera referenciado mediante *M[2]*, cuyo resultado es un valor escalar, concretamente el segundo elemento del vector M.

- Si M es una matriz de 4 dimensiones, cuántas dimensiones tendrán los resultados de las siguientes operaciones?
    - `M[:, :, :, :]`
    - `M[1, 2, 3, 4]`
    - `M[1, 2, 3, :]`
    - `M[:, [1, 2, 3], 4, :]`
    - `M[:, [1, 2, 3], :, :]`
    - `M[[3, 2], 2, 4, :]`
    - `M[[3, 2], [2], [4], [1]]`
    - `M[1, [1, 2, 3], 4, :]`

Existen varias formas de referenciar elementos y partes de matrices. Una de ellas es la ya descrita: utilizar valores enteros o vectores de valores enteros. Otra posibilidad muy utilizada es mediante vectores de valores booleanos. En este caso, se devolverá un objeto que contenga los valores correspondientes del vector a *true*. Si se realiza de esta manera, la longitud del vector debe ser igual al tamaño de la dimensión donde se aplica, por ejemplo:

```
julia> M = [1 2; 3 4; 5 6];
julia> M[ [true,true,false], :]
2×2 Array{Int64,2}:
 1  2
 3  4
julia> M[ :, [true, false] ]
3×1 Array{Int64,2}:
 1
 3
 5
```

También es posible referenciar elementos indicando una matriz de la misma dimensionalidad y del mismo tamaño, pero que contenga valores booleanos, uno por cada elemento de la matriz original. En este caso, se devolverá el resultado como un vector. Por ejemplo, para tomar de la matriz anterior los elementos de valor mayor que 3, esto se puede realizar de la siguiente manera:

```
julia> M[M.>3]
3-element Array{Int64,1}:
 5
 4
 6
```

Existen otras formas de referenciar partes de matrices, como por ejemplo mediante el tipo *CartesianIndex*, que permite referenciar elementos en una posición determinada, pero las dos formas anteriores son las más utilizadas con diferencia.

Para transponer matrices, se puede usar la función *transpose*, o utilizar el operador ’:

```
M'
```

Como resultado, el objeto que devuelve no es de tipo *Array*, sino un tipo más complicado que encapsula a un *Array*. Julia realiza esto para no tener que reservar memoria para el nuevo array traspuesto, sino sencillamente referenciar sus elementos en otro orden, aprovechando la memoria que ya está asignada. Este objeto, a pesar de no ser de tipo *Array*, es un subtipo de *AbstractArray*, por lo que se puede utilizar indistintamente como si fuera un *Array*.

Como se indicó anteriormente, Julia permite aplicar una función a todos los elementos de una matriz y además realizar operaciones entre los elementos de matrices situados en la misma posición. Para esto último, se antepone un punto (.) a operadores como multiplicación, división o potencia:

```
N = [10 20 30; 40 50 60];
N*M
N.*M
N./M
N/10
N>30
N.>30
```

En este último ejemplo, se crea una matriz de valores booleanos en la que cada elemento es el resultado de comprar el elemento correspondiente de la primera matriz con el valor 30.

- ¿Por qué da error al intentar ejecutar N*M pero no N.*M? ¿Por qué da error al intentar ejecutar N>30 pero no N.>30?

Al contrario que en Matlab, Julia no permite que las matrices crezcan dinámicamente. Por ejemplo, en Matlab en la matriz anterior es posible dar valor un elemento situado en una posición que no existe. En este caso, no dará error, sino que Matlab aumentará el tamaño de la matriz hasta esa posición, rellenando los valores nuevos con ceros. En Julia en cambio esto sí provocará un error, puesto que las matrices tienen tamaños definidos.

Una funcionalidad adicional de realizar *broadcast* de operaciones entre arrays es que no es necesario que sean únicamente entre arrays del mismo tamaño o arrays y valores escalares, sino que se puede hacer entre arrays de la misma dimensionalidad pero con distintos tamaños. Esto es muy utilizado en la operación con matrices bidimensionales, y en esta asignatura se utilizará para normalizar los datos. Si se hace un broadcast de una operación entre una matriz (bidimensional) y otra bidimensional con una sola columna, ambas con el mismo número de filas, esta operación se realizará como si la segunda matriz tuviese el mismo tamaño que la primera, con la columna repetida. Lo mismo ocurre si la segunda matriz es una matriz con una fila con el mismo número de columnas. Por supuesto, es totalmente indiferente cuál se pone antes en la operación. A continuación se muestran varios ejemplos sencillos:

```
julia> M = [1 2; 3 4; 5 6; 7 8];
```

```
julia> M.+[1 3]
4×2 Array{Int64,2}:
 2   5
 4   7
 6   9
 8  11
julia> M.+[4; 3; 2; 1]
4×2 Array{Int64,2}:
 5  6
 6  7
 7  8
 8  9
julia> [4; 3; 2; 1].+M
4×2 Array{Int64,2}:
 5  6
 6  7
 7  8
 8  9
julia> M.+[4, 3, 2, 1]
4×2 Array{Int64,2}:
 5  6
 6  7
 7  8
 8  9
```

Como se puede ver en el último ejemplo, si la matriz fila o la matriz columna es un vector, este se tratará como una matriz columna.

Para declarar una matriz y por tanto reservar memoria para la misma se puede usar el tipo de esa matriz como si fuera una función, poniendo como primer argumento la forma de inicializar los datos, y después un número por cada dimensión. La forma de inicializar los datos más común es usar *undef*, que indica que no se quieren inicializar los valores de la matriz (tendrán los valores que hubiera en memoria en ese momento). Por ejemplo, la siguiente llamada crea una matriz bidimensional de 10 filas y 6 columnas:

```
M = Array{Float32,2}(undef, 10, 6)
```

- ¿Por qué dará error escribir *M = Array{Float32,2}(undef, 10)*?

- ¿Cuál será el resultado de las siguientes llamadas?

    - `typeof(Array{Float32,2}(undef, 4, 15))`
    - `typeof(Array{Float32,2})`
    - `isa(Array{Float32,2}, Array{Float32,2})`
    - `isa(Array{Float32,2}(undef, 4, 10), Array{Float32,2})`

Otra posibilidad es usar las funciones *zeros* o *ones*, que crean matrices del tamaño indicado donde todos los elementos son 0 o 1 respectivamente, por ejemplo:

```
zeros(10, 6)
```

```
ones(13)
```

Para concatenar dos vectores, se pueden poner entre corchetes, separando los elementos por ";" Por ejemplo, la siguiente línea permite crear un vector resultado de concatenar los dos vectores indicados:

```
[[1, 2, 3]; [4, 5, 6]]
```

Para unir matrices, esto se realiza de forma similar, utilizando los corchetes, y separando las matrices por espacio, en caso de querer unirlas añadiendo nuevas columnas (poniendo una "al lado" de la otra), o separando las matrices por ";" en caso de querer unirlas añadiendo nuevas filas (poniendo una "debajo" de la otra). Por supuesto, en el primer caso tiene que coincidir el número de filas, y en el segundo tiene que coincidir el número de columnas. Aquí tenéis un ejemplo de ambas situaciones:

```
[[1 2 3; 4 5 6] [7 8 9; 10 11 12]]
[[1 2 3; 4 5 6]; [7 8 9; 10 11 12]]
```

- ¿Cuál será el resultado de cada una de las dos operaciones anteriores?

- Si *[[1, 2, 3]; [4, 5, 6]]* permite concatenar vectores, ¿cuál será el resultado de las siguientes operaciones? ¿Cuál será el tipo del resultado?

- `[[1, 2, 3]  [4, 5, 6]]`
- `[[1, 2, 3], [4, 5, 6]]`
- `[[1  2  3]  [4  5  6]]`
- `[[1  2  3]; [4  5  6]]`
- `[[1  2  3], [4  5  6]]`

Julia dispone de una serie de funciones para trabajar con matrices tales como ones, zeros, size, length, max, min, minmax, rand, inv, det, sum, etc. Algunas de las más utilizadas son:

- zeros: recibe como parámetros el tamaño de cada dimensión, y crea una matriz de esa dimensionalidad y tamaño, donde todos los elementos son igual a 0.
- ones: recibe como parámetros el tamaño de cada dimensión, y crea una matriz de esa dimensionalidad y tamaño, donde todos los elementos son igual a 1.
- rand: recibe como parámetros el tamaño de cada dimensión, y crea una matriz de esa dimensionalidad y tamaño, donde todos los elementos son valores aleatorios.
- size: recibe como parámetro una matriz y devuelve una tupla con tantos elementos como dimensionalidad tenga la matriz, donde cada elemento es el tamaño de esa matriz. Puede ser llamada indicando como segundo argumento la dimensión, y devuelve únicamente el tamaño de esa dimensión.
- maximum: recibe como parámetro una matriz y devuelve el valor máximo de la matriz. Opcionalmente, se puede indicar la dimensión mediante la palabra clave *dims* para que la función se aplique en esa dimensión.
- minimum: realiza la operación similar a máximum, pero devolviendo el valor mínimo en lugar del máximo. Acepta también la palabra clave *dims*.
- findall: recibe como parámetro una matriz de valores booleanos y devuelve los índices de los valores positivos.
- sum: recibe como parámetro una matriz y la suma de los valores de la matriz. Acepta también la palabra clave *dims*.
- mean: recibe como parámetro una matriz y devuelve el valor promedio de la matriz. Acepta también la palabra clave *dims*. Esta función necesita que previamente se haya cargado el paquete *Statistics*.
- std: recibe como parámetro una matriz y devuelve la desviación típica de los valores de la matriz. Acepta también la palabra clave *dims*. Esta función necesita que previamente se haya cargado el paquete *Statistics*

Como se indica, muchas de estas funciones aceptan la palabra clave *dims* para indicar a lo largo de qué dimensión se ejecutará la operación indicada. Si no se utiliza, se realiza esta operación en todos los elementos de la matriz. Si se indica *dims=1*, esta operación se realiza de forma paralela en cada columna, y como resultado se devolverá una matriz de una fila y tantas columnas como tenía la matriz original, donde el valor de cada columna será el resultado de aplicar la función en los elementos de esa columna de la matriz original. Si se indica *dims=2*, esta operación se realiza de forma paralela en cada fila, y como resultado se devolverá una matriz de una columna y tantas filas como tenía la matriz original, donde el valor de cada fila será el resultado de aplicar la función en los elementos de esa fila de la matriz original. A continuación se muestran varios ejemplos:

```
julia> sum([1 2; 3 4])
10
julia> sum([1 2; 3 4], dims=1)
1×2 Array{Int64,2}:
 4  6
julia> sum([1 2; 3 4], dims=2)
2×1 Array{Int64,2}:
 3
 7
```

En general, para ver la ayuda de una función, simplemente escribiendo "?" en el intérprete de comandos se puede escribir el nombre de una función para ver su documentación.

Como ya se ha dicho, Julia permite hacer un *broadcast* de funciones sobre todos los elementos de una matriz, teniendo como resultado una matriz de la misma dimensionalidad, con los resultados de las operaciones. Por ejemplo, llamar a la función cos.(M) devolverá una matriz con los cosenos de cada elemento de la matriz M. La expresión M.>3 devolverá una matriz binaria, del mismo tamaño que M, con unos en aquellos elementos en los que el elemento de M fuese mayor que 3, y ceros en el resto. La expresión M[1,:].=3 asigna el valor 3 a la primera fila de la matriz M.