

¿Venenosa o no?

INDICE

# **Descripción del proyecto**

El propósito de este proyecto es aplicar diferentes algoritmos de aprendizaje automático a un conjunto de datos encontrado en la siguiente URL:

<https://www.kaggle.com/uciml/mushroom-classification>

Estos datos corresponden a un conjunto de atributos de diferentes setas y, en base a estos, si son venenosas o no.

El conjunto de datos presenta un total de 8124 instancias con 22 atributos cada una, y una columna indicando si la seta es venenosa o no.

Cada uno de los datos representa lo explicado en el Excel adjunto.

# **Consideraciones previas**

Como podemos observar en el conjunto de datos original tenemos un conjunto de datos que maneja caracteres para definir el valor de cada atributo. Para una mayor facilidad de cómputo transformaremos dichos atributos a dígitos numéricos según el Excel adjunto. Nos ayudaremos de ciertas expresiones regulares similares a:

sed -i -E 's/^([0-9]+\,[0-9]+\,[0-9]+\,[0-9]+\,)t/\11/g' mushrooms.csv

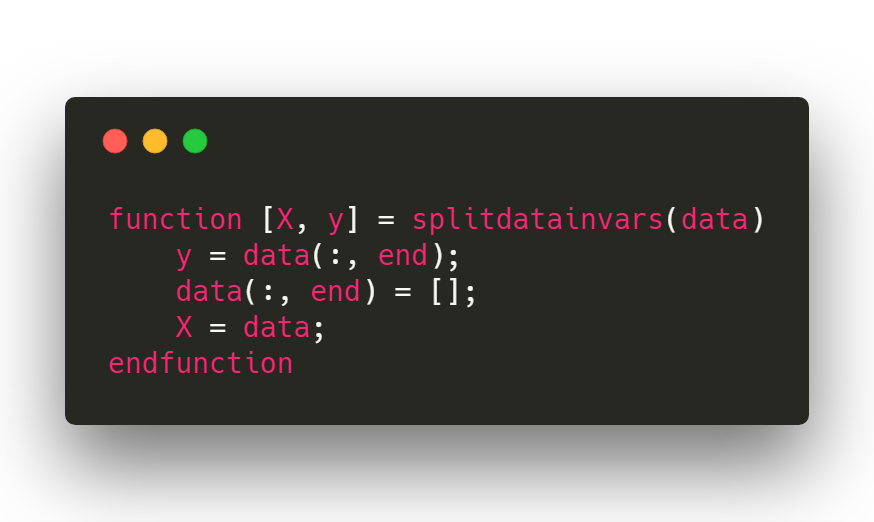
Para adaptar la expresión regular, si queremos sustituir el atributo n repetiremos “[0-9]+\,” n-1 veces y después del cierre de paréntesis escribiremos el carácter que queremos sustituir seguido de “/\1” y el dígito por el que lo sustituiremos.

En el caso de la expresión regular del ejemplo se sustituirá, en la columna del 5º atributo, todos los caracteres t por el dígito 1.

Había datos con columnas con interrogación, por lo que los hemos eliminado antes de empezar con los algoritmos, quedando un total de 5644 instancias.

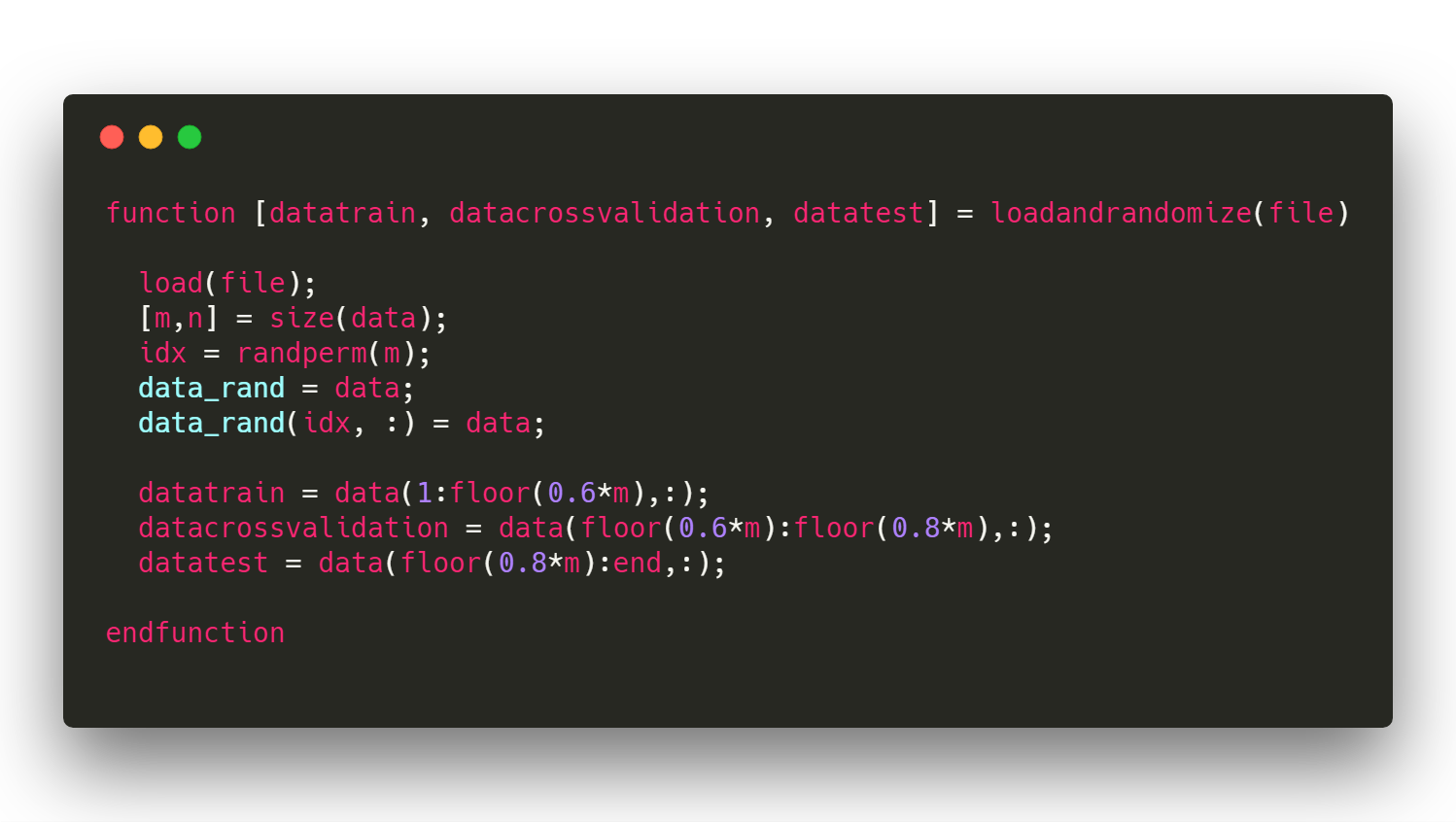
A continuación, crearemos un archivo llamado *mushroomdata* en el que se guardarán los datos de manera que ejecutando la instrucción load se cargarán automáticamente en la variable *data*.

Además, hemos creado un método llamado *splitdatainvars.m*, que al pasarle la variable *data*, devuelve los datos desglosados en X (todos los datos con sus atributos) e Y (clasificación de cada dato). Este es el código:



También hemos creado una función llamada *loadanrandomize.m*, que recibe un nombre de fichero y carga los datos y los separa en 3 subconjuntos aleatoriamente: subconjunto de entrenamiento (60% de los datos), subconjunto de cross-validation y subconjunto de test (20% de los datos respectivamente).

El código de dicho método es el siguiente:



Una vez adaptados los datos de entrada y creados los métodos de carga, hemos aplicado distintas técnicas de aprendizaje automático con distintos detalles y ejecuciones propios para comprobar diferencias tanto en tiempo de ejecución, eficiencia, y porcentaje de datos clasificados correctamente en cada una. Hemos aplicado regresión logística, redes neuronales y Support Vector Machines (SVM)

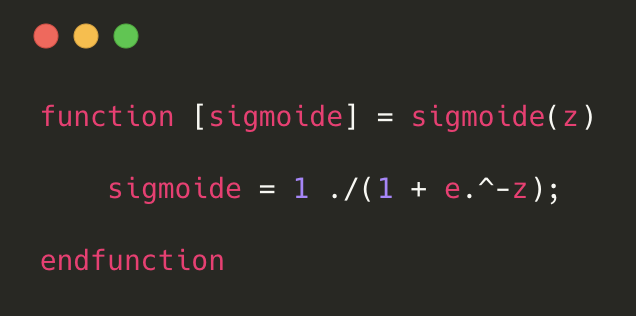
# **Regresión Logística**

El primer algoritmo de aprendizaje empleado ha sido la regresión logística. No ha sido necesario emplear la clasificación multiclase debido a que nuestros datos únicamente tienen dos clases posibles (Venenoso y no venenoso).

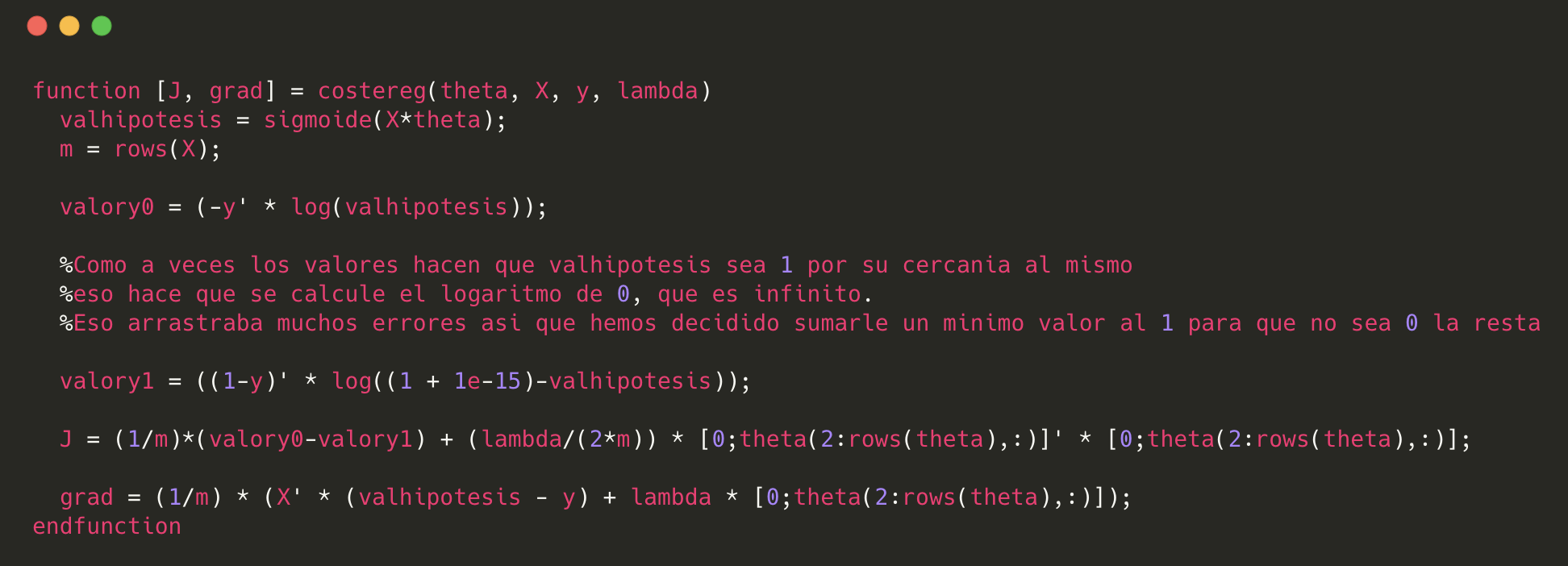
Todo el proceso que se realiza en la regresión logística se encuentra en la función *reglogistica.m*, que, al ser bastante larga debido a todas las pruebas hechas, se irá desglosando por partes.

Se incluye una explicación con lo realizado, los resultados obtenidos y las conclusiones que se han sacado en base a estos resultados.

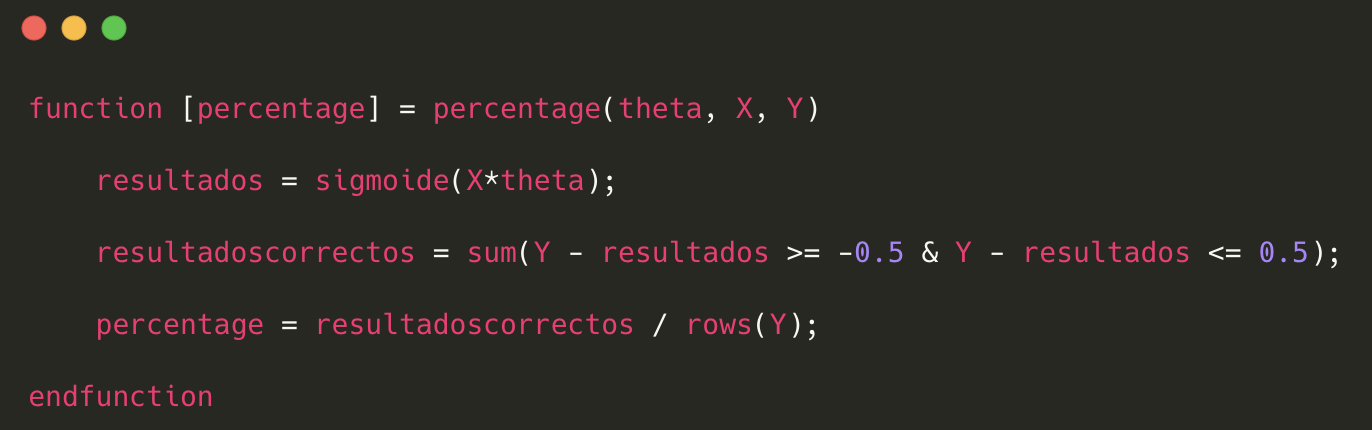
En primer lugar, hemos cargado los datos los hemos separado con la función *splitdatainvars.m* mencionada anteriormente. Además, hemos creado y usado la función *sigmoide.m,* que se encarga de calcular el sigmoide de los datos pasados por parámetro. Este es el código:



A continuación, hemos aplicado regresión logística sobre el 100% de los datos, sin hacer una división entre datos de entrenamiento, de cross-validation y de test en esta ocasión, y con un valor de lambda para la regularización de 0.01, usando la función *fminunc.m.* Para realizar esta llamada se ha necesitado también la función *costereg.m*, que se encarga de calcular el valor de la función de coste y el gradiente de unas thetas dadas. Este es el código:



Las thetas óptimas se han obtenido con un valor de la función de coste J = 0.027619 y un tiempo de ejecución de 0.36 segundos. Posteriormente, hemos usado estas thetas para calcular el porcentaje de los datos clasificados correctamente, usando para ello la función *percentage.m,* que se encarga de comprobar si la clasificación obtenida por el algoritmo de aprendizaje es la misma que la que tiene en los datos, y devolver el porcentaje de aciertos que la regresión logística ha obtenido. Este es el código de esta función:



El resultado obtenido es que, tras realizar la regresión, un 99.82% de los datos de entrenamiento son clasificados correctamente.

Esta es la parte del código de la función principal dedicado a esto:



Después, hemos vuelto a cargar los datos, pero esta vez sí hemos dividido los datos en tres subconjuntos (entrenamiento, cross-validation y test), usando la función *loadanrandomize.m* mencionada anteriormente.

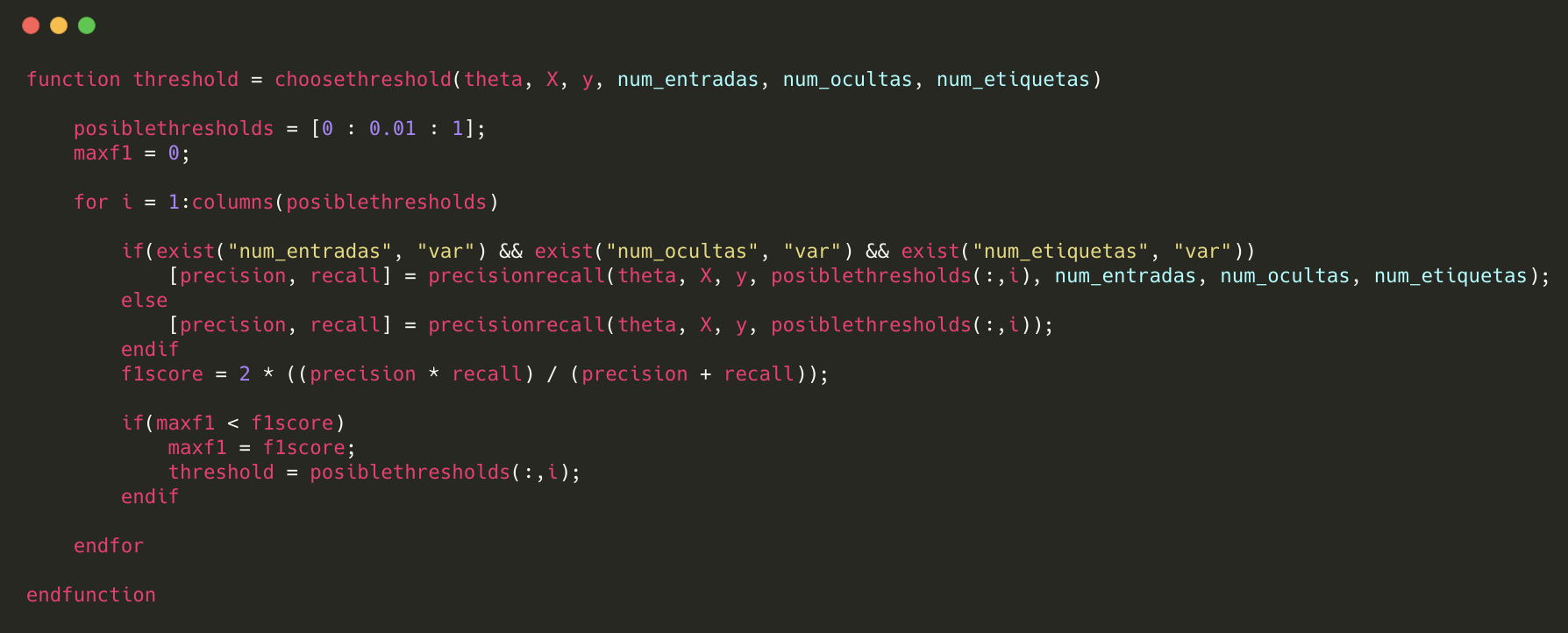
De esta manera, hemos vuelto a aplicar regresión logística, pero esta vez únicamente sobre el subconjunto de entrenamiento, usando el subconjunto de cross-validation para comprobar cuál era la lambda óptima para ser empleado. Para ello, hemos realizado en múltiples ocasiones la regresión logística variando lambda, y calculando el porcentaje de datos de cross-validation que clasificaba correctamente con la función *percentage.m,* y obteniendo el máximo. De esta manera obtenemos que el lamba óptimo es 0.01, clasificando correctamente un 98.05% de los datos de cross-validation. Por último, hemos usado el subconjunto de test para comprobar cuántos datos clasificaba correctamente, obteniendo un porcentaje de 67.96% de datos bien clasificados.

A continuación, se muestra la parte del código de la función principal dedicada a lo explicado anteriormente:

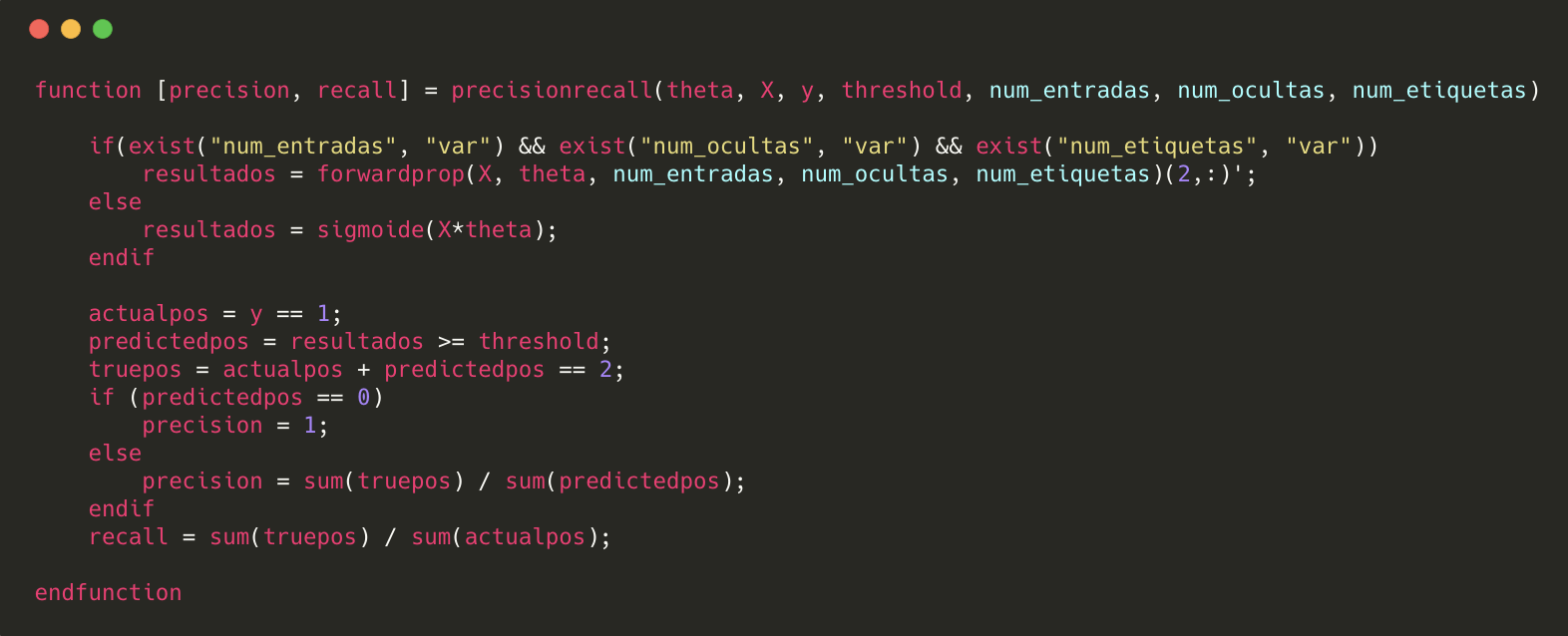


Para finalizar con la regresión logística, hemos calculado el precision y el recall del algoritmo anterior. Hemos usado para ello la función *choosethreshold.m,* que obtiene el mejor threshold de este algoritmo. Se calcula para cada posible valor entre 0 y 1 el F1Score, quedándonos con el mejor. Posteriormente, hemos llamado a la función *precisionrecall.m,* que calcula los valores de precision y recall obtenidos en base a las thetas resultantes del algoritmo de aprendizaje. Para esta llamada no es necesario pasar todos los parámetros.

Este es el código de la función *choosethreshold.m*:

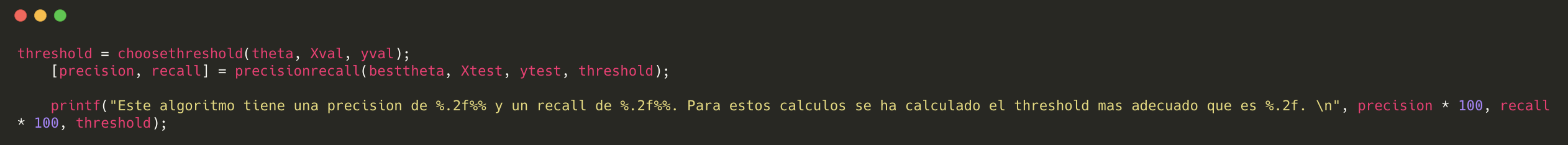


Y este, el código de la función *precisionrecall.m*:

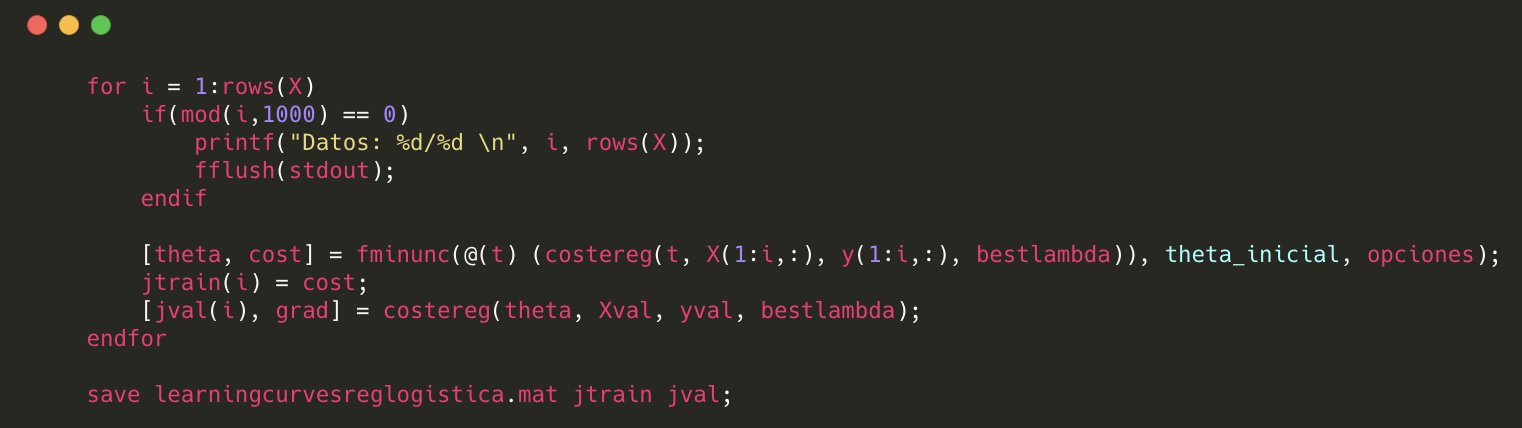


Tras aplicar estas funciones, obtenemos que el threshold óptimo es 0.13, y de esta manera, obteniendo un precision y un recall de 88.97% y 73.60%., respectivamente.

Este es el código de la función principal que se dedica a esta parte:



Por último hemos dibujado las curvas de aprendizaje y de variación de lambda, el siguiente código guarda los valores de J para más tarde dibujar dichas curvas:



Dibujando las curvas de aprendizaje obtenemos esta gráfica:

Y la siguiente gráfica será el resultado de la evolución de los costes al variar el valor de lambda:

**CONCLUSIONES**

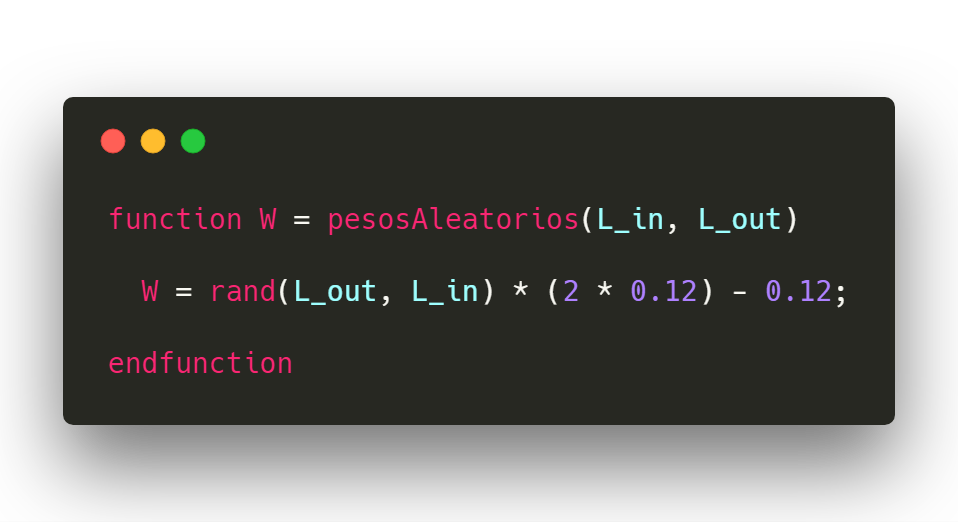
Como conclusiones tras aplicar la regresión logística de esta manera, podemos destacar la diferencia de porcentaje de acierto en la clasificación al dividir los datos en subconjuntos frente a no hacerlo (siendo mayor en el segundo caso). Sin embargo, esto hace que funcione perfecto para los datos de entrenamiento, pero no tan bien para la predicción de nuevos datos. Por ello, es mejor separar los datos, comprobar cuál es la mejor regularización probando distintas lambdas y ver su porcentaje real de aciertos usando datos independientes de los datos usados para entrenar el algoritmo. También podemos apreciar que, a pesar de ser mejor, tarda mucho mas tiempo el primer caso que el segundo.

**Redes Neuronales**

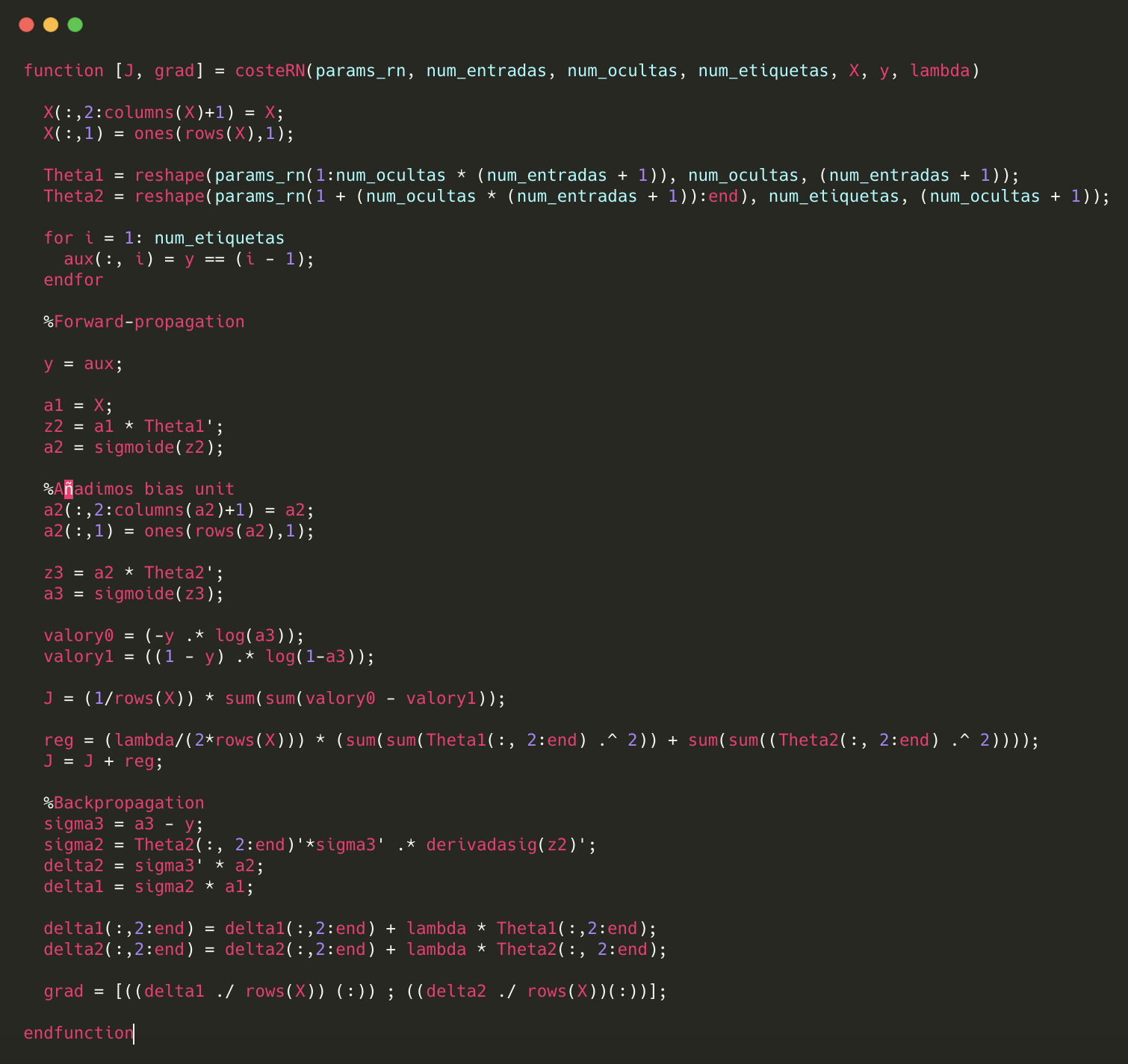
El siguiente algoritmo de aprendizaje empleado ha sido las redes neuronales. Para ello, todo el proceso se ha realizado en una función *neuralnetwork.m,* que al igual que en el caso anterior, dividiremos en partes explicando el proceso.

Para comenzar, hemos cargado los datos en las variables y hemos decidido aplicar varias veces el algoritmo cambiando el número de neuronas de la capa oculta (usando una red neuronal de tres capas), para ver los distintos resultados.

En primer lugar, hemos creado una red neuronal con únicamente tres neuronas en la capa oculta, sin desglosar los datos de entrada en los tres subconjuntos. Primero hemos creado un array que luego será desglosado en las distintas thetas de la red neuronal (en este caso, al haber tres capas, habrá dos subconjuntos de thetas) usando la función *pesosAleatorios.m*, cuyo código es el siguiente:

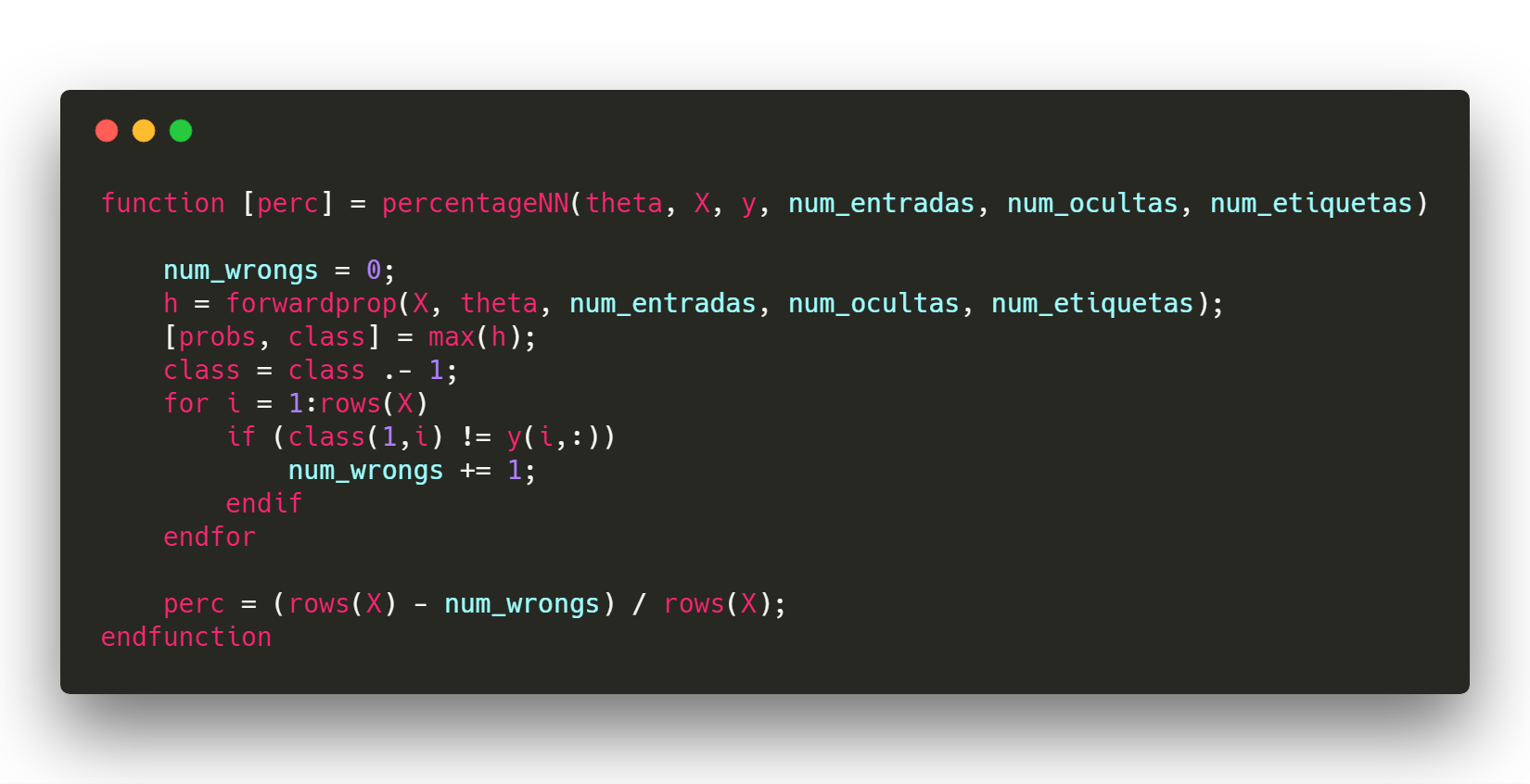


Una vez dado formato a los datos, creadas las thetas iniciales, y elegido un valor de lambda para la regularización (en este caso, 0.01), hemos aplicado las redes neuronales llamando a la función *fmincg.m.* Para realizar la llamada, se ha necesitado además la función *costeRN.m*, que se encarga de calcular el valor de la función de coste y el gradiente para unas thetas dadas. Este es el código:



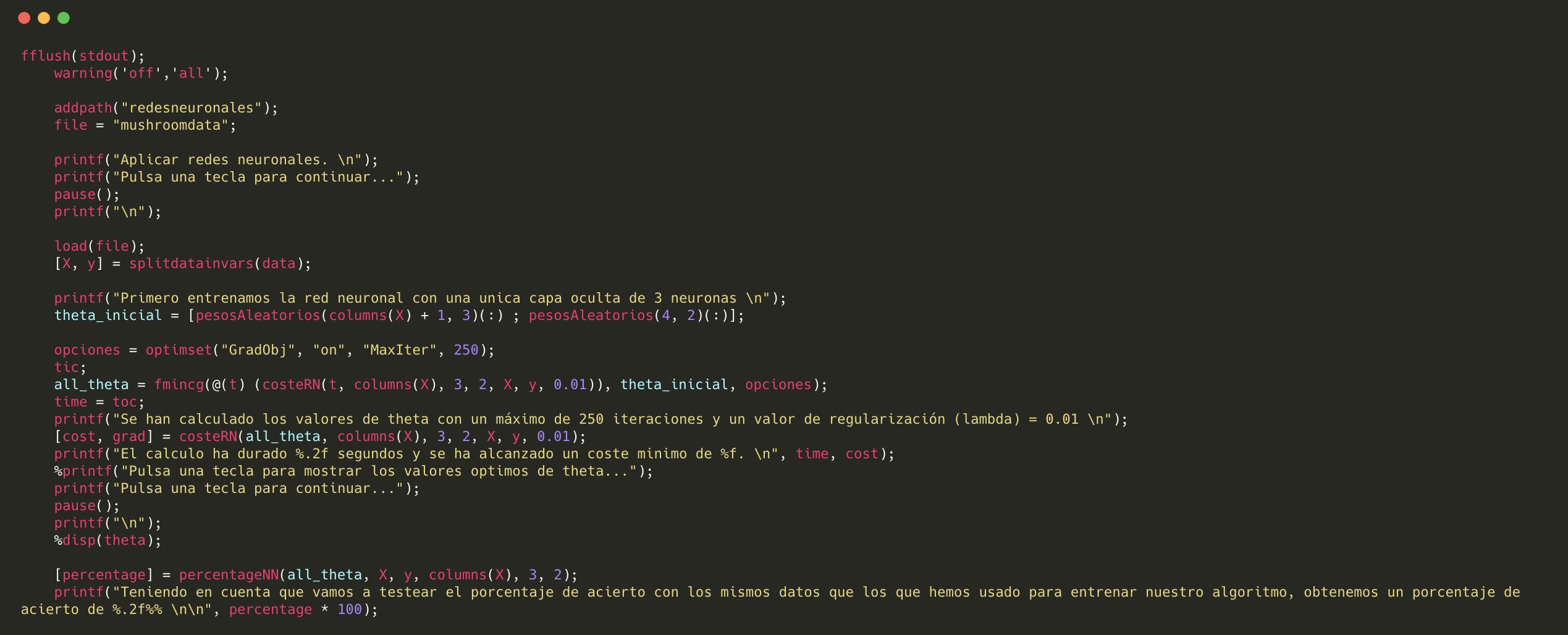
Tras su ejecución, que ha durado 3.85 segundos, se han obtenido las thetas óptimas con un valor de la función de coste J = 0.114869.

Posteriormente, se ha calculado el porcentaje que datos que clasifica correctamente. En este caso, usando la función *percentageNN.m,* cuyo código es el siguiente:



En este caso concreto, el porcentaje de aciertos ha sido de un 98.30% (este porcentaje se ha obtenido al valorar los mismos datos usados para entrenar la red neuronal)

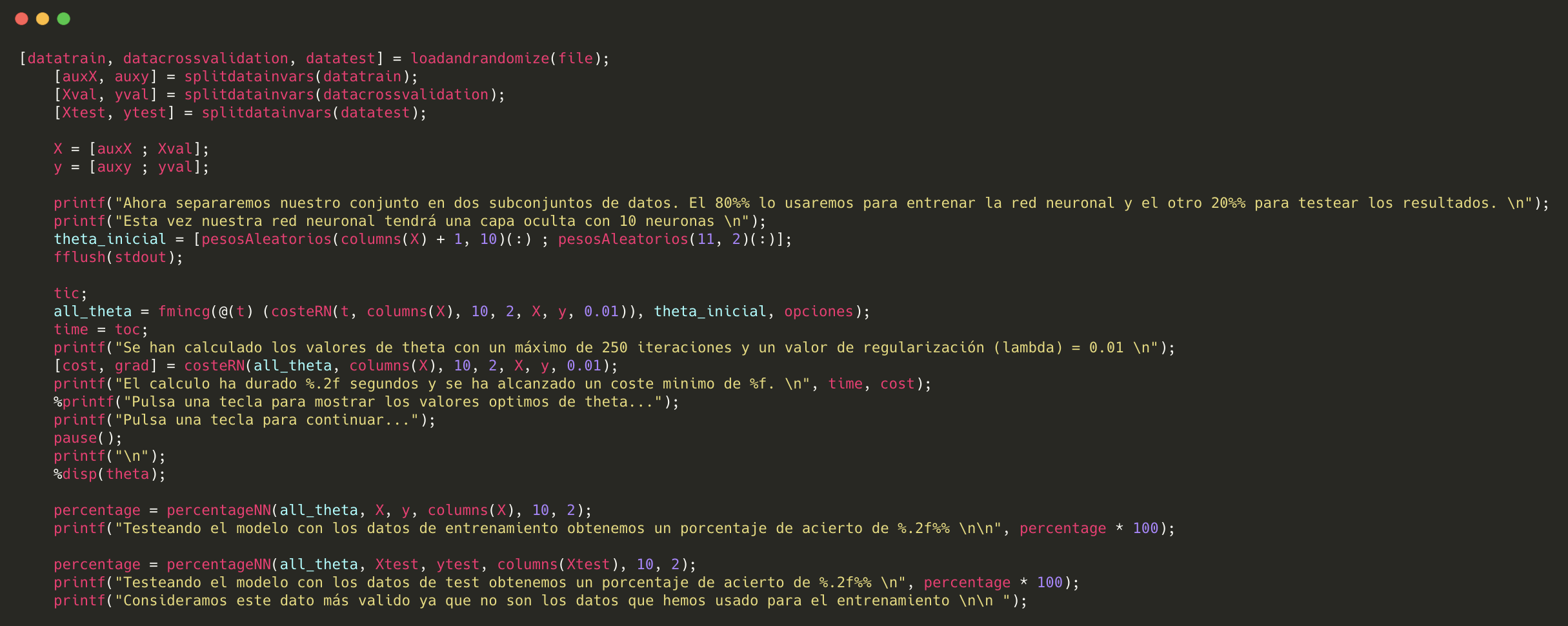
Este es el código de la función principal que se ocupa de esta parte:



En segundo lugar, hemos aplicado una red neuronal con tres capas, pero esta vez con diez neuronas en la capa oculta. Además, hemos decidido usar el 80% de los datos para entrenar la red neuronal y el 20% restante para realizar el test. Para ello, se han seguido los mismos pasos que en el caso anterior, únicamente modificando el array de las thetas iniciales con la nueva dimensión. El valor de lamba para la regularización sigue siendo de 0.01.

Tras aplicar el entrenamiento de la red neuronal, en esta ocasión ha durado 6.01 segundos, y se alcanza un coste mínimo de la función de coste J = 0.002540. Tras calcular el porcentaje de acierto tanto de los datos de entrenamiento como de los de test con la función *percentageNN.m* mencionada anteriormente, clasifica correctamente 100.00% y 94.16%, respectivamente. Consideramos más valido el segundo porcentaje ya que esos datos no se han usado para entrenar la red neuronal.

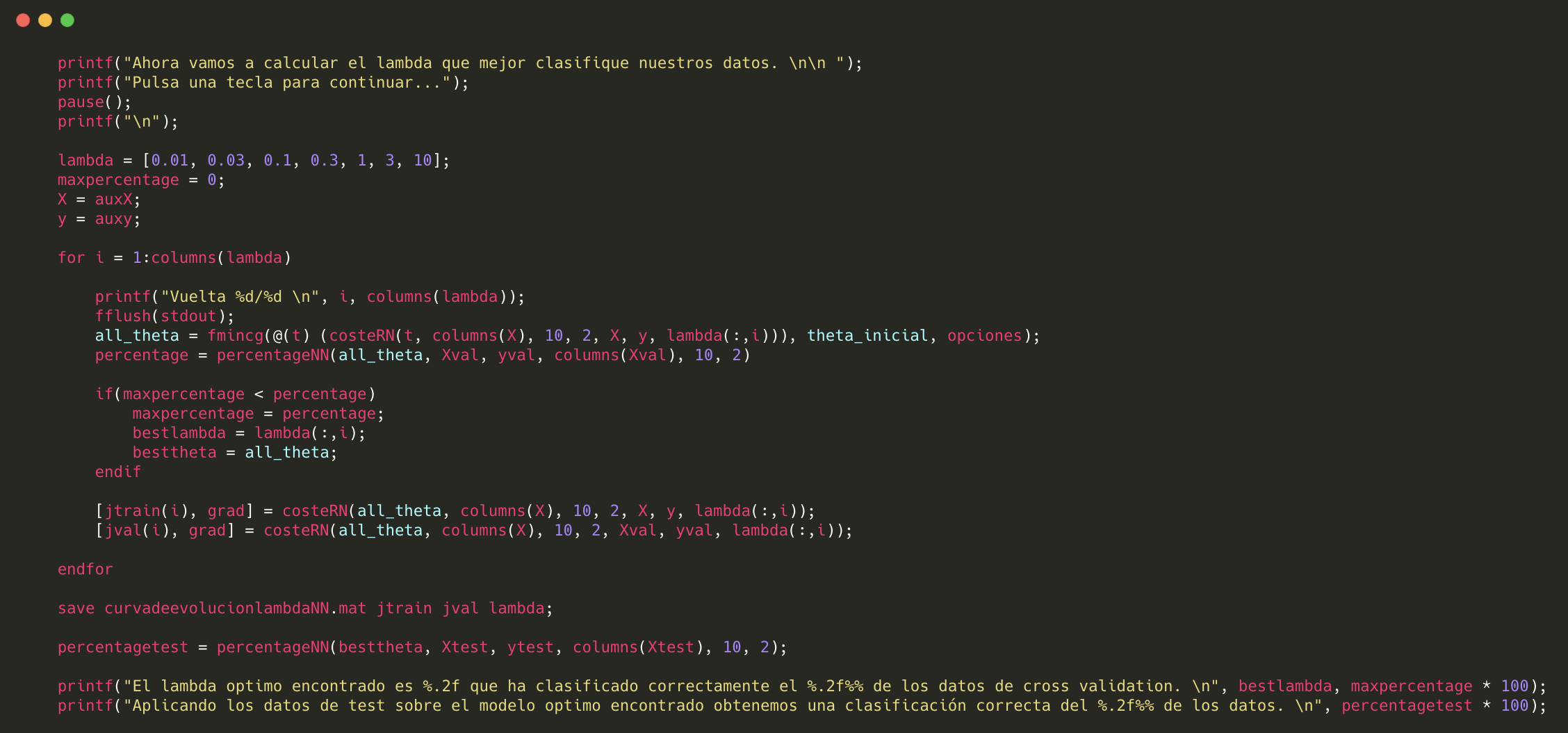
Este es el código de esta segunda parte en la función principal:



En tercer lugar, se ha aplicado una tercera red neuronal, con las mismas dimensiones que la anterior, pero en este caso, dividiendo los datos en tres subconjuntos, siendo estos los datos de entrenamiento, los de cross-validation y los de test. De esta forma, hemos empleado los datos de cross-validation para determinar cuál es el mejor valor de lambda para la regularización. Tras aplicar esta parte, el valor óptimo de lambda es 0.03. Con este valor de lambda clasifica correctamente un 98.14% de los datos de cross-validation.

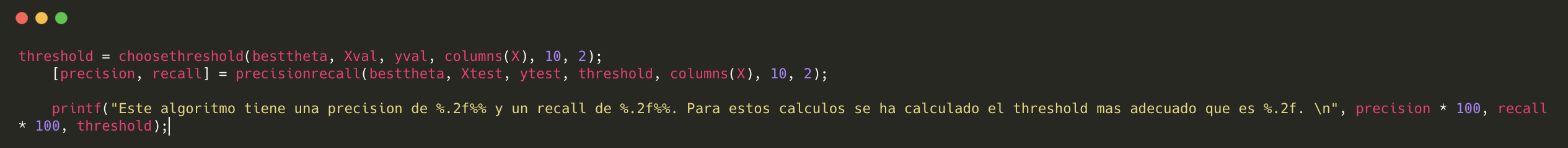
Al aplicar la red neuronal con este valor de lambda, y al comprobar el porcentaje de datos que clasifica correctamente (del subconjunto de test), se obtiene un porcentaje del 69.29%.

Este es el código de la tercera parte de la función principal:

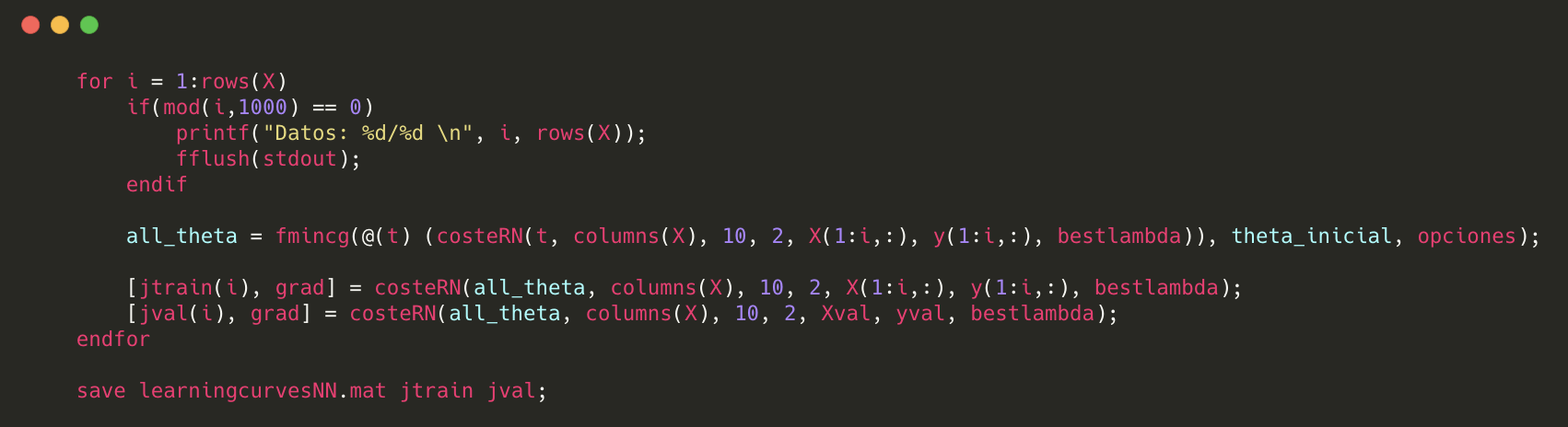


Por último, hemos calculado el precision y el recall del algoritmo anterior. Hemos usado para ello la función *choosethreshold.m,* que obtiene el mejor threshold de este algoritmo. Se calcula de manera similar a la regresión logística, aunque cambiando un poco el código para adecuarlo, quedándonos con el mejor. Posteriormente, hemos llamado a la función *precisionrecall.m,* que calcula los valores de precision y recall obtenidos en base a las thetas resultantes del algoritmo de aprendizaje. Estas funciones tienen el mismo código que en la regresión logística, pero ahora se le pasan más variables.

Tras la ejecución, se ha obtenido un precision de 91.64% y un recall de 100.00% , siendo el threshold 0.01. El código para obtener estos datos es el siguiente:



También hemos guardado los costes al ir añadiendo datos para obtener la curva de aprendizaje con el siguiente fragmento de código:



Dibujando las curvas de aprendizaje obtenemos esta gráfica:

Y la siguiente gráfica será el resultado de la evolución de los costes al variar el valor de lambda:

**CONCLUSIONES**

Tras aplicar varias veces la función *neuralnetwork.m,* podemos obtener una serie de conclusiones. Se observa que la red neuronal funciona mejor cuando se dedican más datos a entrenar la red neuronal, habiendo una gran diferencia al usar el 100% de los datos, el 80% y el 60%, respectivamente.

Además, referente a la conclusión anterior, como la lambda óptima obtenido es 0.01, al usar el 80% de los datos para entrenar la red neuronal, se obtiene un porcentaje de acierto del 89.82%, siendo bastante superior al obtenido por la regresión logística con los mismos datos (un 67.43% en este caso).

También se puede apreciar una gran diferencia al usar el 60% de los datos o el 80% para entrenar, ya que en el primer caso se obtiene un porcentaje de tan solo 65.84%. No hay tanta diferencia al usar el 100%, aunque sería mayor (pero en este caso, no hay datos de test suficientes por lo que podría darse overfitting).

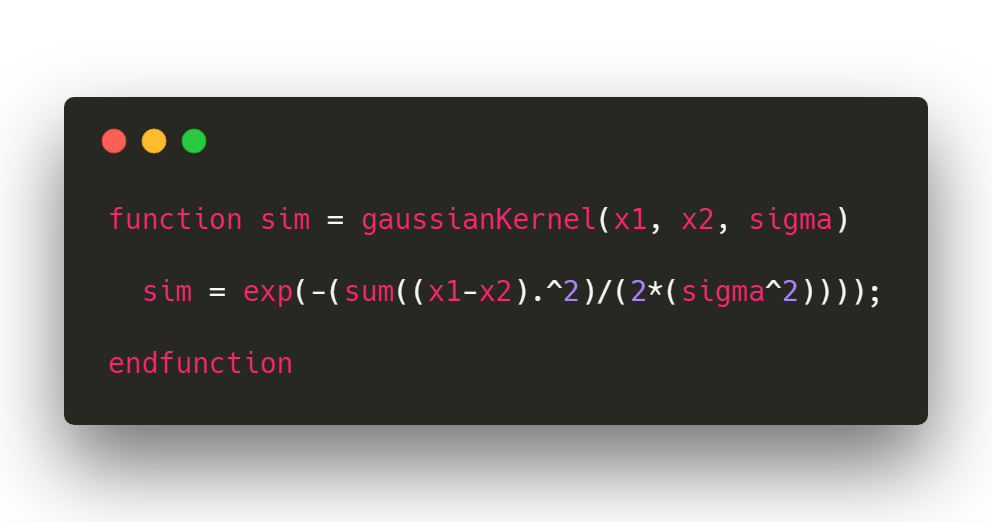
Además, teniendo en cuenta que en estos datos es más importante que el mayor número de setas venenosas sean clasificadas como tal, es mejor reducir el threshold para que, aunque se sacrifique eficacia total, y se clasifiquen algunas no venenosas como nocivas, se asegure que la mayoría de setas venenosas no sean clasificadas como inocuas. (poner lambda = 0.01 y threshold = 0.3 y 0.5 para ver como varia)

**Support Vector Machines (SVM)**

El siguiente algoritmo de aprendizaje empleado para este proyecto ha sido el SVM. Para ello, al igual que en casos anteriores, hemos realizado distintos cálculos para obtener distintos resultados y sacar conclusiones. La función principal de este apartado se llama *svm.m*.

En primer lugar, hemos ejecutado el algoritmo con un kernel gaussiano (en la función *gaussianKernel.m*), con un valor de C (un parámetro de regularización) de 1.

El código de la función es el siguiente:



250.37 segs / 100% / 67.96%

45.01 segs / 100% / 90.71%

C y sigma = 0.01 y 0.1 / 99.56% (cv) / 90.71%(test)

Hay que hacer comentarios a lo largo de la práctica sobre el bias y la variance