

¿Venenosa o no?

# INDICE

Descripción del proyecto	3
Consideraciones previas	3
Regresión Logística	5
Redes Neuronales	13
Support Vector Machines	21
Tabla comparativa de los 3 algoritmos	24

## 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

Los cálculos y tiempos de este proyecto se han tomado usando un Macbook Pro con las siguientes especificaciones:

- Procesador: Intel Core i5 2,3GHz
- Memoria RAM: 8GB LPDDR3 2133MHz
- GPU: Intel Iris Plus Graphics 640 1536 MB

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.

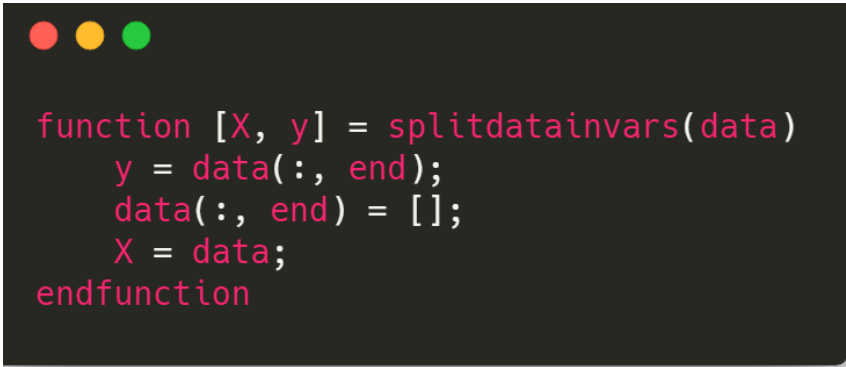
Había datos con columnas con interrogación, por lo que los hemos eliminado antes de empezar con los algoritmos, quedando un total de 5644 instancias.

Como la columna que indicaba el veil-type era constante y siempre era 1 hemos decidido eliminarla.

A continuación, crearemos un archivo llamado *mushroomdata* en el que se guardarán los datos preparados para ser cargados 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:



```
function [X, y] = splitdatainvars(data)
    y = data(:, end);
    data(:, end) = [];
    X = data;
endfunction
```

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:

```

function [datatrain, datacrossvalidation, datatest] = loadandrandomize(file)

    load(file);
    [m,n] = size(data);
    idx = randperm(m);
    data_rand = data;
    data_rand(idx, :) = data;

    datatrain = data(1:floor(0.6*m),:);
    datacrossvalidation = data(floor(0.6*m):floor(0.8*m),:);
    datatest = data(floor(0.8*m):end,:);

endfunction

```

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 el de 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*.

A continuación, iremos explicando las distintas partes de esta función.

En primer lugar, hemos cargado los datos y los hemos separado con la función *splitdatainvars.m*.

Después hemos normalizado los valores dividiendo cada columna entre el máximo de esta. Para eso hemos creado esta función:

```
function [X_norm, maxim] = featureNormalize(X)

    maxim = max (X);
    X_norm = X ./ maxim;

end
```

Hemos realizado la normalización de esta manera en lugar de restando la media y dividiendo entre la desviación estándar porque con el segundo método, al ser valores discretos y a veces tener una desviación muy pequeña, al dividir nuestro conjunto en subconjuntos la desviación podía ser 0. Siendo la desviación 0, la normalización devolvía valores no deseados (NaN o Infinity) y causaba fallos.

A continuación, hemos aplicado regresión logística sobre el 100% de los datos, sin hacer una división de los datos 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 a partir de unos valores de X, y, lambda y theta. Este es el código:

```
function [J, grad] = costereg(theta, X, y, lambda)
    valhipotesis = sigmoide(X*theta);
    m = rows(X);

    valory0 = (-y' * log(valhipotesis));
    valory1 = ((1 - y)' * log(1 - valhipotesis));

    J = (1/m)*(valory0+valory1) + (lambda/(2*m)) * [0;theta(2:rows(theta),:)]' * [0;theta(2:rows(theta),:));

    grad = (1/m) * (X' * (valhipotesis - y) + lambda * [0;theta(2:rows(theta),:)]);
endfunction
```

Para el calculo de la sigmoide hemos utilizado la siguiente función:

```
function [sigmoide] = sigmoide(z)
    sigmoide = 1 ./ (1 + e.^-z);
endfunction
```

Una vez acabado el descenso de gradiente se ha obtenido un valor mínimo de  $J = 0.085527$ . El tiempo de ejecución necesario ha sido de 0.64 segundos.

Posteriormente, hemos usado las thetas calculadas para obtener el porcentaje de datos clasificados correctamente, usando para ello la función *percentage.m*. Este es el código de la función:

```
function [percentage] = percentage(theta, X, Y)
    resultados = sigmoide(X*theta);
    resultadoscorrectos = sum(Y - resultados >= -0.5 & Y - resultados <= 0.5);
    percentage = resultadoscorrectos / rows(Y);
endfunction
```

El resultado obtenido es que, tras realizar la regresión, un 98.09% de los datos de entrenamiento son clasificados correctamente.

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

```

function reglogistica()
    fflush(stdout);
    warning('off','all');

    addpath("reglogistica");
    file = "mushroomdata";

    printf("Aplicar regresión logística regularizada. \n");
    printf("Pulsa una tecla para continuar...");
    pause();
    printf("\n");

    load(file);
    [X, y] = splitdatainvars(data);

    theta_inicial = zeros(columns(X) + 1, 1);
    [newX, maxm] = featureNormalize(X);

    newX(:, 2:columns(newX) + 1) = newX;
    newX(:, 1) = ones(rows(newX), 1);

    opciones = optimset("GradObj", "on", "MaxIter", 500);
    tic;
    [theta, cost] = fminunc(@(t) (costereg(t, newX, y, 0.01)), theta_inicial, opciones);
    time = toc;
    fflush(stdout);
    printf("Se han calculado los valores de theta con un máximo de 500 iteraciones y un valor de regularización (lambda) = 0.01 \n");
    printf("El calculo ha durado %.2f segundos y se ha alcanzado un coste mínimo de %f. \n", time, cost);
    %printf("Pulsa una tecla para mostrar los valores optimos de theta...");
    printf("Pulsa una tecla para continuar...");
    pause();
    fflush(stdout);
    printf("\n");
    %disp(theta);

    [percentage] = percentage(theta, newX, y);
    printf("Teniendo en cuenta que vamos a testear el porcentaje de acierto con los mismos datos que los que hemos usado para entrenar nuestro algoritmo, obtenemos un porcentaje de acierto de %.2f%% \n\n", percentage * 100);

```

Después, hemos vuelto a cargar los datos, pero esta vez los hemos dividido en tres subconjuntos (entrenamiento, cross-validation y test), usando la función *loadandrandomize.m* que ejecuta el siguiente código:

```

function [datatrain, datacrossvalidation, datatest] = loadandrandomize(file)

    load(file);
    [m,n] = size(data);
    idx = randperm(m);
    data_rand = data;
    data_rand(idx, :) = data;

    datatrain = data(1:floor(0.6*m),:);
    datacrossvalidation = data(floor(0.6*m):floor(0.8*m),:);
    datatest = data(floor(0.8*m):end,:);

endfunction

```

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 y el subconjunto de test para dar unos resultados con unos datos diferentes.

Para ello, hemos realizado el descenso de gradiente variando lambda, y calculando el porcentaje de datos de cross-validation que clasificaba correctamente con la función *percentage.m*. De esta manera obtenemos que el lambda que mayor porcentaje de



datos del subconjunto cross-validation clasifica es 0.01, obteniendo un coste de 0.0056585 y clasificando correctamente un 98.05% de los datos. Por último, hemos usado el subconjunto de test para comprobar cuántos datos (desconocidos para el algoritmo hasta el momento) clasificaba correctamente, obteniendo un porcentaje de 66.37% de datos bien clasificados.

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

```

[datatrain, datacrossvalidation, datatest] = loadandrandomize(file);
[X, y] = splitdatainvars(datatrain);
[Xval, yval] = splitdatainvars(datacrossvalidation);
[Xtest, ytest] = splitdatainvars(datatest);

[newX, maxm] = featureNormalize(X);

newXval = Xval ./ maxm;

newXtest = Xtest ./ maxm;

newX(:, 2:columns(newX) + 1) = newX;
newX(:, 1) = ones(rows(newX), 1);

newXval(:, 2:columns(newXval) + 1) = newXval;
newXval(:, 1) = ones(rows(newXval), 1);

newXtest(:, 2:columns(newXtest) + 1) = newXtest;
newXtest(:, 1) = ones(rows(newXtest), 1);

printf("Ahora aplicaremos regresión logística regularizada separando nuestro set de datos en 3 subconjuntos (train, cv y test). \n");
printf("Pulsa una tecla para continuar...");
pause();
printf("\n");

lambda = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30];
maxpercentage = 0;

for i = 1:columns(lambda)

    printf("Vuelta %d/%d \n", i, columns(lambda));
    fflush(stdout);
    [theta, cost] = fminunc(@(t) (costereg(t, newX, y, lambda(:,i))), theta_inicial, opciones);
    percentageval = percentage(theta, newXval, yval);

    if(maxpercentage < percentageval)
        maxpercentage = percentageval;
        bestlambda = lambda(:,i);
        besttheta = theta;
        bestcost = cost;
    endif

    jtrain(i) = cost;
    [jval(i), grad] = costereg(theta, newXval, yval, lambda(:,i));

endfor

save curvadeevolucionlambdareglogistica.mat jtrain jval lambda;

[percentagetest] = percentage(besttheta, newXtest, ytest);
printf("El lambda optimo encontrado es %.2f que ha clasificado correctamente el %.2f%% de los datos de cross validation. \n", bestlambda, maxpercentage * 100);
printf("Aplicando los datos de test sobre el modelo optimo encontrado obtenemos una clasificación correcta del %.2f%% de los datos. \n", percentagetest * 100);

```

Para disponer de más resultados hemos calculado la precisión y el recall de nuestro algoritmo.

Hemos usado para ello la función *choosethreshold.m*, que obtiene el mejor threshold de este algoritmo. Para cada posible valor entre 0 y 1, calculamos el  $F_1$ Score, quedándonos con el threshold que genere el mayor  $F_1$ Score.

Posteriormente, hemos llamado a la función *precisionrecall.m*, que calcula los valores de precisión y recall obtenidos en base a las thetas resultantes del algoritmo de aprendizaje y con el threshold calculado en el paso anterior.

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

```

function threshold = choosethreshold(theta, X, y, num_entradas, num_ocultas, num_etiquetas)

    posiblethresholds = [0 : 0.01 : 1];
    maxf1 = 0;

    for i = 1:columns(posiblethresholds)

        if(exist("num_entradas", "var") && exist("num_ocultas", "var") && exist("num_etiquetas", "var"))
            [precision, recall] = precisionrecall(theta, X, y, posiblethresholds(:,i), num_entradas, num_ocultas, num_etiquetas);
        else
            [precision, recall] = precisionrecall(theta, X, y, posiblethresholds(:,i));
        endif
        f1score = 2 * ((precision * recall) / (precision + recall));

        if(maxf1 < f1score)
            maxf1 = f1score;
            threshold = posiblethresholds(:,i);
        endif

    endfor

endfunction

```

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

```

function [precision, recall] = precisionrecall(theta, X, y, threshold, num_entradas, num_ocultas, num_etiquetas)

    if(exist("num_entradas", "var") && exist("num_ocultas", "var") && exist("num_etiquetas", "var"))
        resultados = forwardprop(X, theta, num_entradas, num_ocultas, num_etiquetas)(2,:);
    else
        resultados = sigmoide(X*theta);
    endif

    actualpos = y == 1;
    predictedpos = resultados >= threshold;
    truepos = actualpos + predictedpos == 2;
    if (predictedpos == 0)
        precision = 1;
    else
        precision = sum(truepos) / sum(predictedpos);
    endif
    recall = sum(truepos) / sum(actualpos);

endfunction

```

Tras estos pasos, obtenemos que el threshold óptimo es 0.11 (que resulta en un F1Score de 0.3982) y con este threshold obtenemos una precisión y un recall de 88.78% y 72.23% respectivamente.

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

```

threshold = choosethreshold(theta, newXval, yval);
[precision, recall] = precisionrecall(besttheta, newXtest, ytest, threshold);

printf("Este algoritmo tiene una precision de %.2f%% y un recall de %.2f%%. Para estos calculos se ha calculado el threshold mas adecuado que es %.2f. \n", precision * 100, recall * 100, threshold);

```

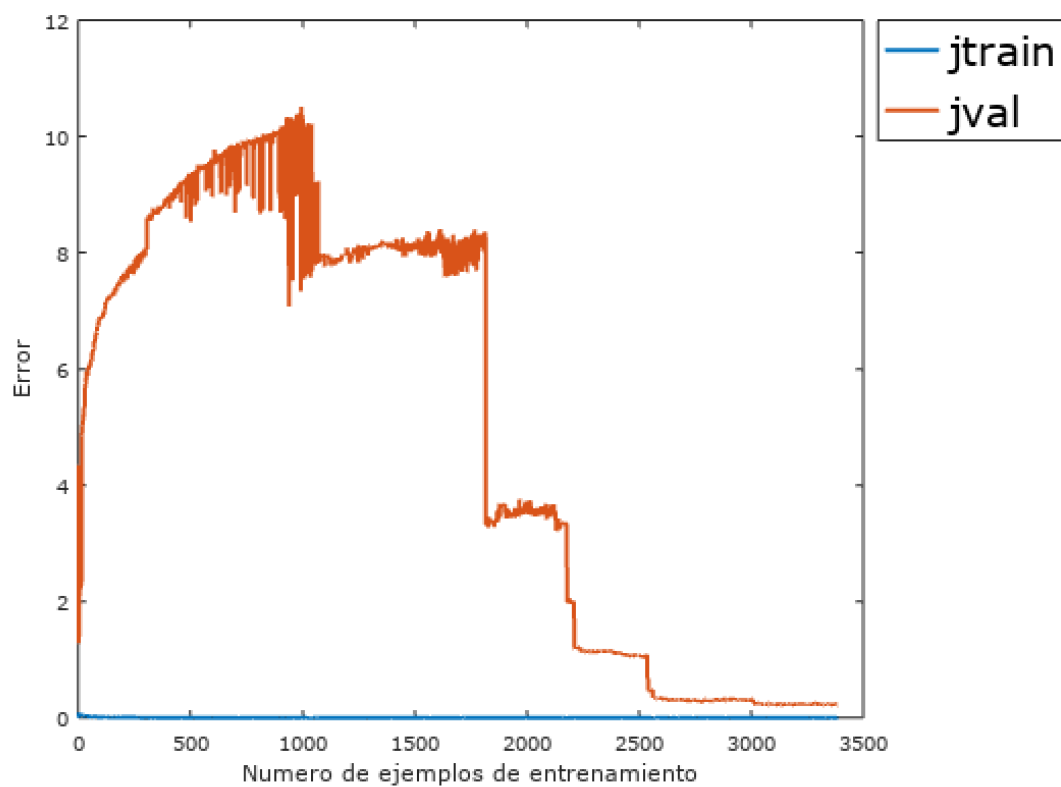
Por último, hemos dibujado las curvas de aprendizaje y de variación de lambda, el siguiente código guarda los valores de J para más tarde dibujar la curva de aprendizaje (los valores para la curva de variación de lambda se guardan durante el cálculo de esta):

```
for i = 1:rows(newX)
    if(mod(i,1000) == 0)
        printf("Datos: %d/%d \n", i, rows(X));
        fflush(stdout);
    endif

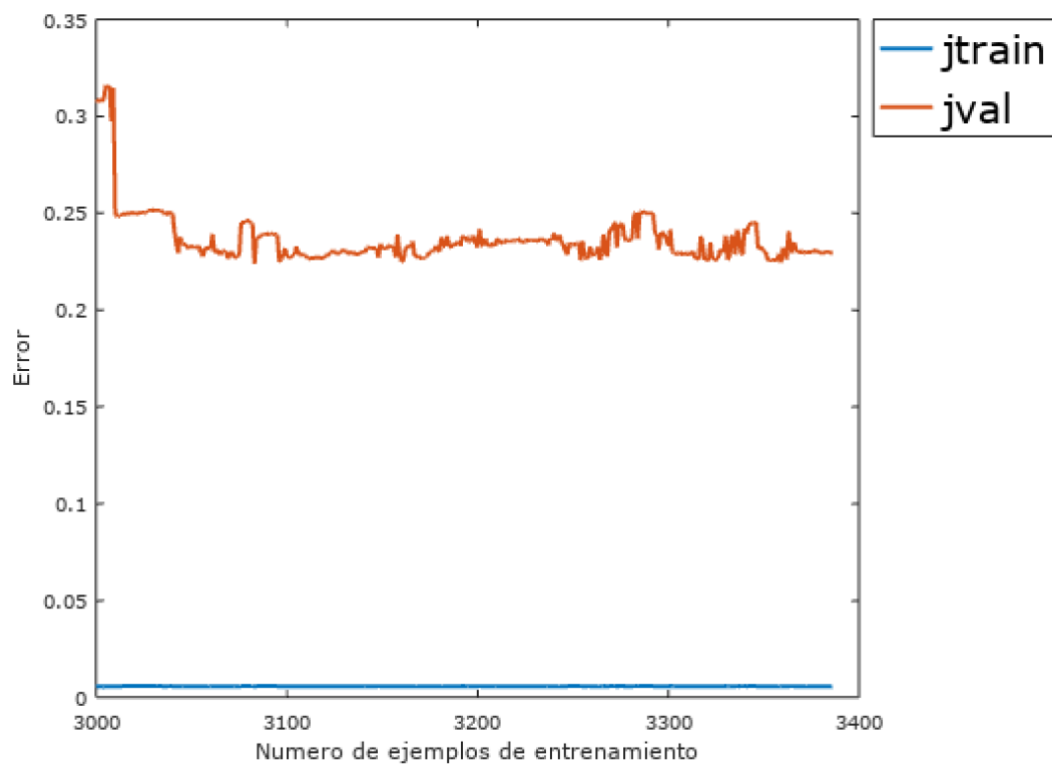
    [theta, cost] = fminunc(@(t) (costereg(t, newX(1:i,:), y(1:i,:), bestlambda)), theta_inicial, opciones);
    jtrain(i) = cost;
    [jval(i), grad] = costereg(theta, newXval, yval, bestlambda);
endfor

save learningcurvesreglogistica.mat jtrain jval;
```

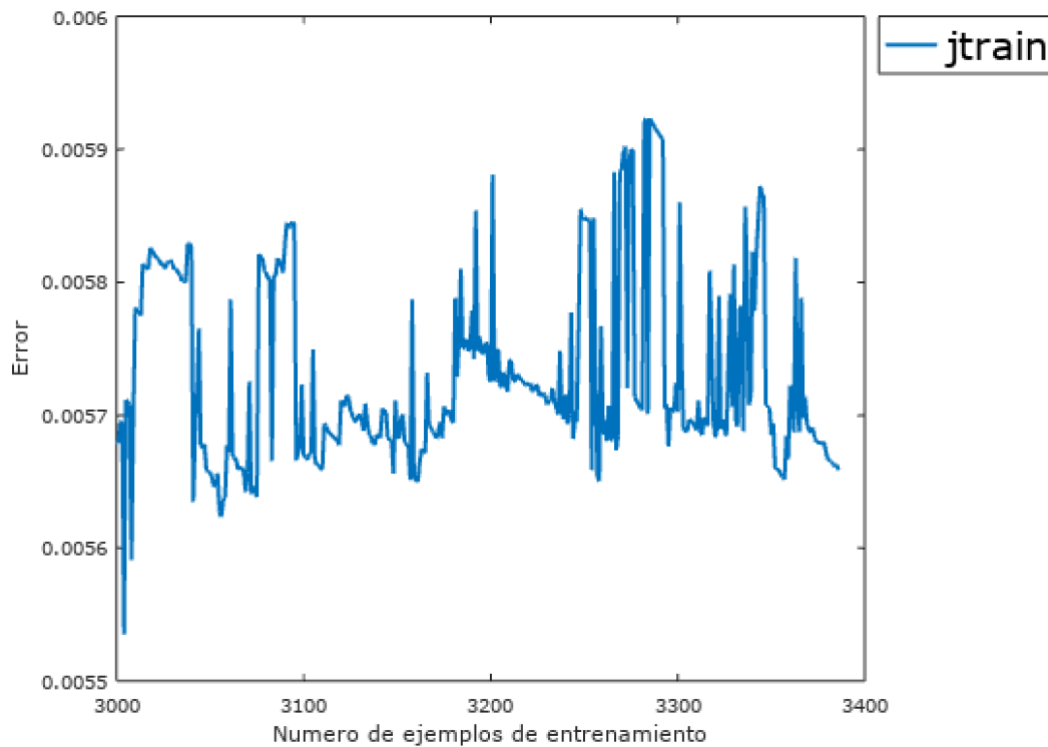
Estas serían las curvas de aprendizaje obtenidas:



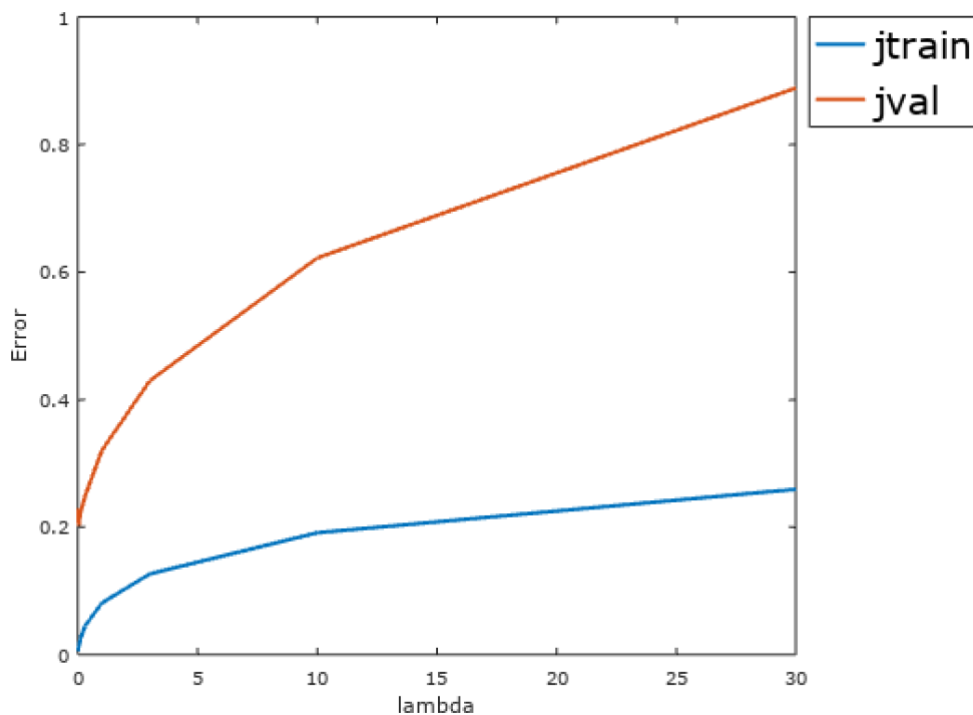
Mostrando solo desde los 3000 valores hasta el final (para observar más detalle) vemos que la gráfica se muestra así:



Aun así, seguimos sin poder ver el detalle de la evolución de valores de la función de coste para los datos de entrenamiento y podría parecer que no varía, por lo que hemos decidido mostrar solo esos valores para que se aprecie la variación (ya que varía en un rango de valores muy pequeño en comparación con  $j_{val}$ ):



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. Como vemos el porcentaje de acierto es mayor en el caso de no dividir nuestro subconjunto, pero esto no nos ayuda a decidir si nuestro algoritmo es un buen clasificador, ya que lo único que indica es que clasifica bien los datos con los que se ha entrenado, pero no datos desconocidos para él. 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. Podemos apreciar que, a pesar de ser mejor, tarda mucho más tiempo el primer caso que el segundo.

También cabe destacar el hecho de que el valor de la función de coste para el subconjunto de validación se hace menor según vamos aumentando el número de ejemplos de entrenamiento, siendo lo más cercano posible al valor de la función de coste para el subconjunto de entrenamiento cuando alcanzamos el máximo de ejemplos disponible. Deducimos, por tanto, que cuanto mayor sea el conjunto de

datos de entrenamiento menor será la función de coste para los datos de validación, llegando en algún momento a igualar el valor de la función de coste para los datos de entrenamiento.

También podemos observar que el threshold que mejor  $F_1$ Score consigue es demasiado pequeño (0.11). Esto es porque la mayoría de nuestros casos de entrenamiento son negativos (i.e. setas no venenosas). Por este motivo también tenemos un recall tan bajo a pesar de ser el threshold 0.11, ya que le cuesta encontrar un gran número de setas venenosas entre los datos de test. También podemos ver que la precisión es buena y que las setas clasificadas como venenosas, en efecto serán venenosas.

Consideramos que es un threshold demasiado bajo, pero también el adecuado para que la clasificación sea segura, ya que el tema tratado es delicado para la salud. Para poder permitirnos aumentarlo deberíamos disponer de más datos de entrenamiento.



## Redes Neuronales

El siguiente algoritmo de aprendizaje empleado ha sido una red neuronal. Para ello, todo el proceso se ha realizado en una función *neuralnetwork.m*, que explicaremos a continuación.

Hemos decidido aplicar varias veces el algoritmo cambiando el número de neuronas de la capa oculta (usando una red neuronal de una única capa oculta), para ver los distintos resultados.

En primer lugar, hemos creado una red neuronal con únicamente tres neuronas en la capa oculta, usando todos los datos disponibles para el entrenamiento. Hemos inicializado los valores de theta aleatoriamente usando la función *pesosAleatorios.m*, cuyo código es el siguiente:

```
function W = pesosAleatorios(L_in, L_out)

    W = rand(L_out, L_in) * (2 * 0.12) - 0.12;

endfunction
```

A continuación, hemos entrenado la red neuronal con un valor de lambda para la regularización de 0.01. Para realizar el descenso de gradiente hemos utilizado la función *fmincg.m*.

Para el cálculo de la función de coste hemos creado una función llamada *costeRN.m*, que se encarga de calcular el valor de la función de coste y el gradiente para unos valores de theta.

Este es el código de costeRN donde se realiza el forward-propagation y el back-propagation:

```
function [J, grad] = costeRN(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, lambda)

    X(:,2:columns(X)+1) = X;
    X(:,1) = ones(rows(X),1);

    Theta1 = reshape(params_rn(1:num_ocultas * (num_entradas + 1)), num_ocultas, (num_entradas + 1));
    Theta2 = reshape(params_rn(1 + (num_ocultas * (num_entradas + 1)):end), num_etiquetas, (num_ocultas + 1));

    for i = 1: num_etiquetas
        aux(:, i) = y == (i - 1);
    endfor

    %Forward-propagation

    y = aux;

    a1 = X;
    z2 = a1 * Theta1';
    a2 = sigmoide(z2);

    %Añadimos bias unit
    a2(:,2:columns(a2)+1) = a2;
    a2(:,1) = ones(rows(a2),1);

    z3 = a2 * Theta2';
    a3 = sigmoide(z3);

    valory0 = (-y .* log(a3));
    valory1 = ((1 - y) .* log(1-a3));

    J = (1/rows(X)) * sum(sum(valory0 - valory1));

    reg = (lambda/(2*rows(X))) * (sum(sum(Theta1(:, 2:end) .^ 2)) + sum(sum((Theta2(:, 2:end) .^ 2))));
    J = J + reg;

    %Backpropagation
    sigma3 = a3 - y;
    sigma2 = Theta2(:, 2:end)'*sigma3 .* derivadasig(z2)';
    delta2 = sigma3' * a2;
    delta1 = sigma2 * a1;

    delta1(:,2:end) = delta1(:,2:end) + lambda * Theta1(:,2:end);
    delta2(:,2:end) = delta2(:,2:end) + lambda * Theta2(:, 2:end);

    grad = [((delta1 ./ rows(X)) (:) ) ; ((delta2 ./ rows(X))(:))];

endfunction
```

Tras su ejecución, que ha durado 2.29 segundos, se han obtenido las thetas óptimas con un valor mínimo de la función de coste  $J = 0.003449$ .

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

```
function [perc] = percentageNN(theta, X, y, num_entradas, num_ocultas, num_etiquetas)

    num_wrongs = 0;
    h = forwardprop(X, theta, num_entradas, num_ocultas, num_etiquetas);
    [probs, class] = max(h);
    class = class .- 1;
    for i = 1:rows(X)
        if (class(1,i) != y(i,:))
            num_wrongs += 1;
        endif
    endfor

    perc = (rows(X) - num_wrongs) / rows(X);
endfunction
```

En este caso concreto, el porcentaje de aciertos ha sido de un 100.00%. Este porcentaje se ha obtenido al valorar los mismos datos usados para entrenar la red neuronal, por lo que no es un dato fiable sobre la exactitud del algoritmo.

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

```
function neuralnetwork()
    fflush(stdout);
    warning('off','all');

    addpath("redesneuronales");
    file = "mushroomdata";

    printf("Aplicar redes neuronales. \n");
    printf("Pulsa una tecla para continuar...");
    pause();
    printf("\n");

    load(file);
    [X, y] = splitdatainvars(data);
    [newX, maxin] = featureNormalize(X);

    printf("Primero entrenamos la red neuronal con una unica capa oculta de 3 neuronas \n");
    theta_inicial = [pesosAleatorios(columns(newX) + 1, 3)(); pesosAleatorios(4, 2)()];

    opciones = optimset("GradObj", "on", "MaxIter", 250);
    tic;
    all_theta = fmincg(@(t) (costeRN(t, columns(newX), 3, 2, newX, y, 0.01)), theta_inicial, opciones);
    time = toc;
    printf("Se han calculado los valores de theta con un máximo de 250 iteraciones y un valor de regularización (lambda) = 0.01 \n");
    [cost, grad] = costeRN(all_theta, columns(newX), 3, 2, newX, y, 0.01);
    printf("El calculo ha durado %.2f segundos y se ha alcanzado un coste minimo de %. \n", time, cost);
    %printf("Pulsa una tecla para mostrar los valores optimos de theta...");
    printf("Pulsa una tecla para continuar...");
    pause();
    printf("\n");
    %disp(theta);

    [percentage] = percentageNN(all_theta, newX, y, columns(newX), 3, 2);
    printf("Teniendo en cuenta que vamos a testear el porcentaje de acierto con los mismos datos que los que hemos usado para entrenar nuestro algoritmo, obtenemos un porcentaje de acierto de %.2f%% \n\n", percentage * 100);
endfunction
```

En segundo lugar, hemos entrenado la red neuronal 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 lambda para la regularización sigue siendo 0.01.

Tras aplicar el entrenamiento de la red neuronal, en esta ocasión ha durado 4.06 segundos, y se alcanza un coste mínimo de la función de coste  $J = 0.002366$ . 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 93.89% respectivamente. Consideramos más válido 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:

```

[datatrain, datacrossvalidation, datatest] = loadandrandomize(file);
[X, y] = splitdatainvars(datatrain);
[Xval, yval] = splitdatainvars(datacrossvalidation);
[Xtest, ytest] = splitdatainvars(datatest);

[newX, maxIm] = featureNormalize([X ; Xval]);

newXtest = Xtest ./ maxIm;

y = [y ; yval];

printf("Ahora separaremos nuestro conjunto en dos subconjuntos de datos. El 80% lo usaremos para entrenar la red neuronal y el otro 20% para testear los resultados. \n");
printf("Esta vez nuestra red neuronal tendrá una capa oculta con 10 neuronas \n");
theta_inicial = [pesosAleatorios(columns(newX) + 1, 10)(:); pesosAleatorios(11, 2)(:)];
fflush(stdout);

tic;
all_theta = fmincg(@(t) (costeRN(t, columns(newX), 10, 2, newX, y, 0.01)), theta_inicial, opciones);
time = toc;
printf("Se han calculado los valores de theta con un máximo de 250 iteraciones y un valor de regularización (lambda) = 0.01 \n");
[cost, grad] = costeRN(all_theta, columns(newX), 10, 2, newX, y, 0.01);
printf("El calculo ha durado %.2f segundos y se ha alcanzado un coste minimo de %f. \n", time, cost);

printf("Pulsa una tecla para continuar...");
pause();
printf("\n");

percentage = percentageNN(all_theta, newX, y, columns(newX), 10, 2);
printf("Testeando el modelo con los datos de entrenamiento obtenemos un porcentaje de acierto de %.2f%% \n\n", percentage * 100);

percentage = percentageNN(all_theta, newXtest, ytest, columns(newXtest), 10, 2);
printf("Testeando el modelo con los datos de test obtenemos un porcentaje de acierto de %.2f%% \n", percentage * 100);
printf("Consideramos este dato más valido ya que no son los datos que hemos usado para el entrenamiento \n\n ");

```

En tercer lugar, se ha entrenado 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 este entrenamiento obtenemos que el valor óptimo de lambda es 0.10. Con este valor de lambda clasifica correctamente un 98.05% de los datos de cross-validation y el valor de la función de coste es 0.010507.

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 66.46%.

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

```

printf("Ahora vamos a calcular el lambda que mejor clasifique nuestros datos. \n\n ");
printf("Pulsa una tecla para continuar...");
pause();
printf("\n");

clear -x file opciones theta_inicial;

[datatrain, datacrossvalidation, datatest] = loadandrandomize(file);
[X, y] = splitdatainvars(datatrain);
[Xval, yval] = splitdatainvars(datacrossvalidation);
[Xtest, ytest] = splitdatainvars(datatest);

[newX, maxin] = featureNormalize(X);

newXval = Xval ./ maxin;
newXtest = Xtest ./ maxin;

lambda = [0.01, 0.03, 0.1, 0.3, 1, 3, 10];
maxpercentage = 0;
minvalcost = realmax;

for i = 1:columns(lambda)

    printf("Vuelta %d/%d \n", i, columns(lambda));
    fflush(stdout);
    all_theta = fmincg(@(t) (costeRN(t, columns(newX), 10, 2, newX, y, lambda(:,i))), theta_inicial, opciones);
    percentage = percentageNN(all_theta, newXval, yval, columns(newXval), 10, 2);

    [jtrain(i), grad] = costeRN(all_theta, columns(newX), 10, 2, newX, y, lambda(:,i));
    [jval(i), grad] = costeRN(all_theta, columns(newX), 10, 2, newXval, yval, lambda(:,i));

    if(maxpercentage < percentage || (maxpercentage == percentage && jval(i) < minvalcost))
        minvalcost = jval(i);
        maxpercentage = percentage;
        bestlambda = lambda(:,i);
        besttheta = all_theta;
    endif

endfor

save curvadeevolucionlambdaNN.mat jtrain jval lambda;

percentagetest = percentageNN(besttheta, newXtest, ytest, columns(newXtest), 10, 2);

printf("El lambda optimo encontrado es %.2f que ha clasificado correctamente el %.2f%% de los datos de cross validation. \n", bestlambda, maxpercentage * 100);
printf("Aplicando los datos de test sobre el modelo optimo encontrado obtenemos una clasificación correcta del %.2f%% de los datos. \n", percentagetest * 100);

```

Por último, hemos calculado la precisión y el recall de la red neuronal entrenada. Hemos usado para ello la función *choosethreshold.m*, que obtiene el threshold con un mayor  $F_1$ Score. 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 precisión 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 88.97% y un recall de 73.60%, siendo el threshold 0.01 (que resulta en un  $F_1$  Score de 0.4027). El código para obtener estos datos es el siguiente:

```

threshold = choosethreshold(besttheta, Xval, yval, columns(X), 10, 2);
[precision, recall] = precisionrecall(besttheta, Xtest, ytest, threshold, columns(X), 10, 2);

printf("Este algoritmo tiene una precision de %.2f%% y un recall de %.2f%%. Para estos calculos se ha calculado el threshold mas adecuado que es %.2f. \n", precision * 100, recall * 100, threshold);

```

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

```

for i = 1:rows(X)
    if(mod(i,1000) == 0)
        printf("Datos: %d/%d \n", i, rows(X));
        fflush(stdout);
    endif

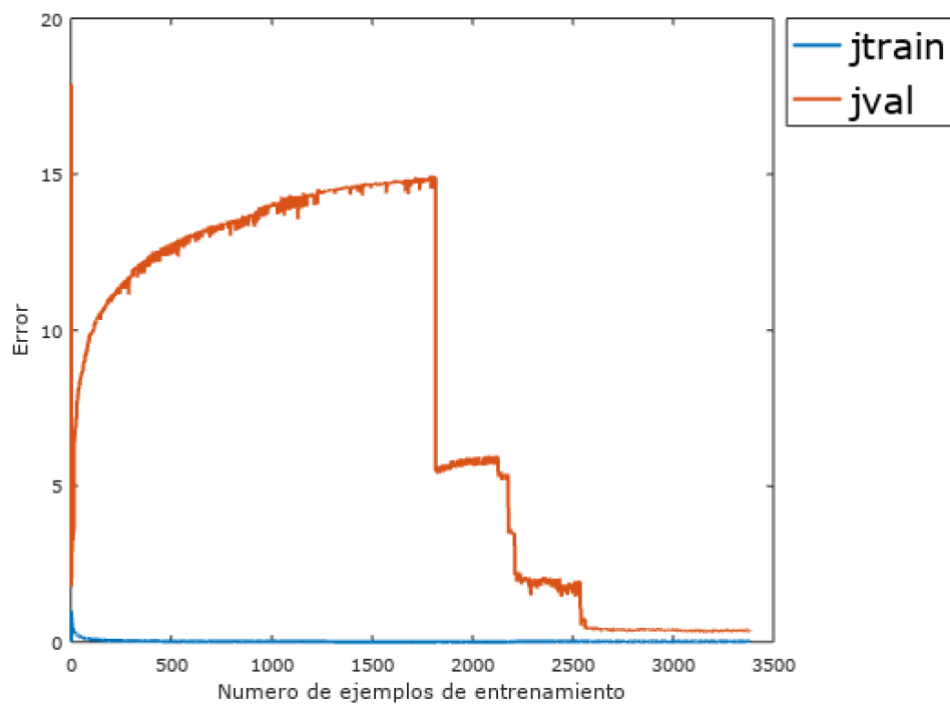
    all_theta = fmincg(@(t) (costeRN(t, columns(X), 10, 2, X(1:i,:), y(1:i,:), bestlambda)), theta_inicial, opciones);

    [jtrain(i), grad] = costeRN(all_theta, columns(X), 10, 2, X(1:i,:), y(1:i,:), bestlambda);
    [jval(i), grad] = costeRN(all_theta, columns(X), 10, 2, Xval, yval, bestlambda);
endfor

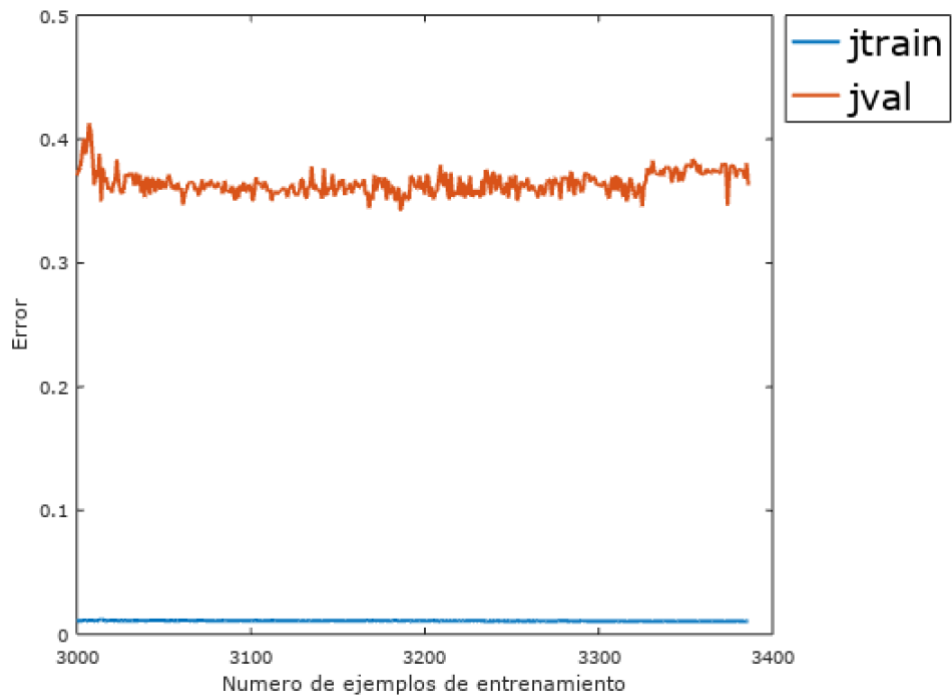
save learningcurvesNN.mat jtrain jval;

```

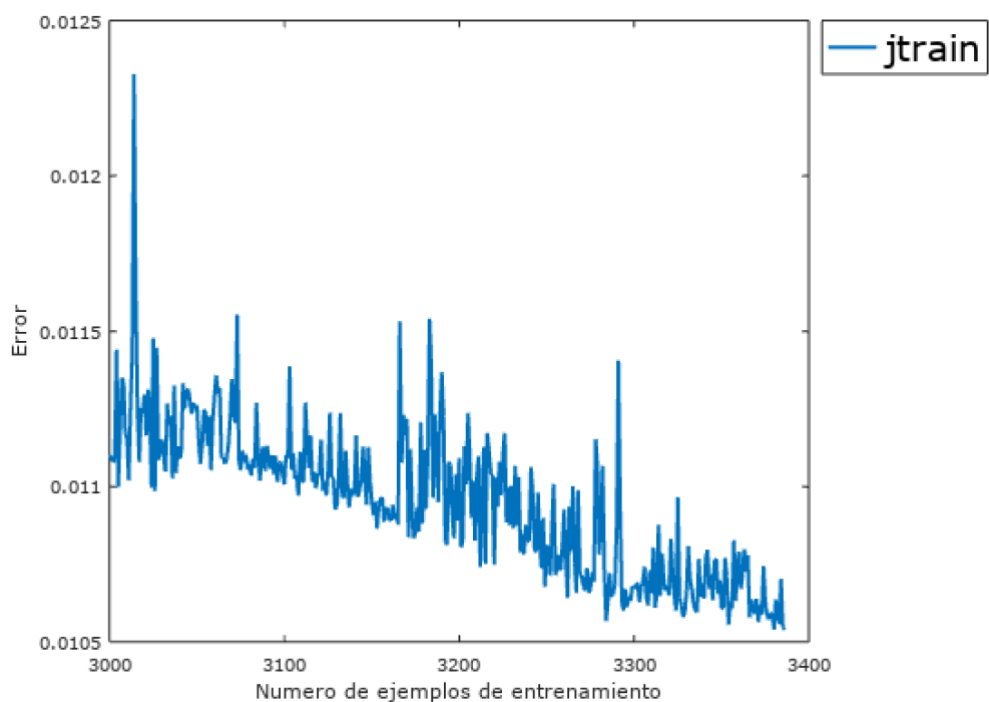
Dibujando las curvas de aprendizaje obtenemos esta gráfica:



Mostrando solo desde los 3000 valores hasta el final (para observar con más detalle cómo se estabiliza el coste al final) vemos que la gráfica se muestra así:

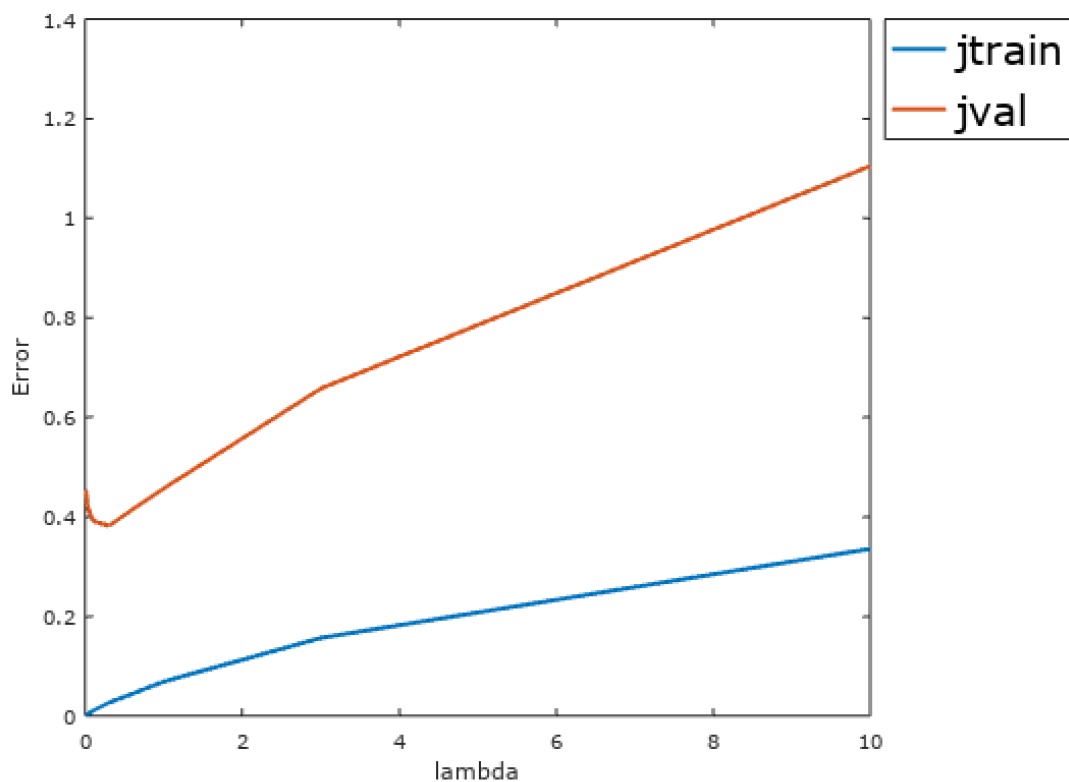


Aun así, seguimos sin poder ver el detalle de la evolución de valores de la función de coste para los datos de entrenamiento y podría parecer que no varía, por lo que hemos decidido mostrar solo esos valores para que se aprecie la variación (ya que varía en un rango de valores muy pequeño en comparación con jval):





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



## CONCLUSIONES

Tras entrenar nuestra red neuronal podemos obtener una serie de conclusiones.

Observando las curvas de aprendizaje podemos notar que la red neuronal disminuye drásticamente su coste (de cross-validation) cuantos más datos usamos para entrenar.

Además, comparándolo con la regresión logística no se obtiene un porcentaje de acierto mucho mayor (tan solo 0.09% más), por lo que en este problema podría no ser adecuado aplicar una red neuronal ya que el beneficio obtenido es demasiado pequeño en comparación con el coste de ejecución, que aumenta significativamente.

Los porcentajes de precisión y recall tampoco aumentan excesivamente, lo que supone otro argumento más en contra de aplicar una red neuronal en este problema.

También podemos observar que el threshold que mejor  $F_1$ Score consigue es demasiado pequeño (0.01). Esto es porque la mayoría de nuestros casos de entrenamiento son negativos (i.e. setas no venenosas). Por este motivo también tenemos un recall tan bajo a pesar de ser el threshold 0.01, ya que le cuesta encontrar un gran número de setas venenosas entre los datos de test. También podemos ver que la precisión es buena y que las setas clasificadas como venenosas, en efecto serán venenosas.

Consideramos que es un threshold demasiado bajo, pero también el adecuado para que la clasificación sea segura, ya que el tema tratado es delicado para la salud. Para poder permitirnos aumentarlo deberíamos disponer de más datos de entrenamiento.

## Support Vector Machines (SVM)

El siguiente algoritmo de aprendizaje empleado para este proyecto ha sido el SVM.

La función principal de este apartado se llama *svm.m* y la desarrollaremos a continuación.

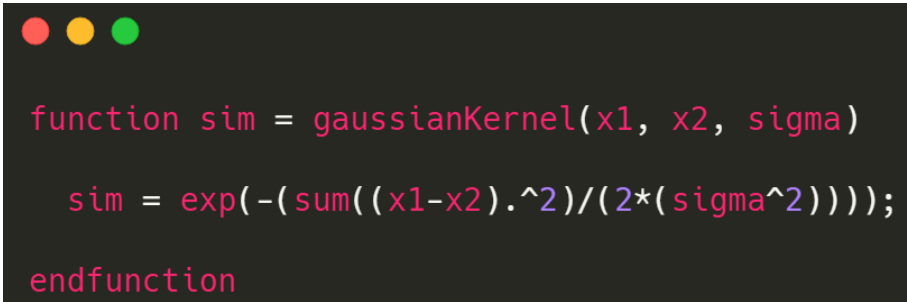
Para este algoritmo hemos separado directamente los datos en 3 subconjuntos.

En primer lugar, hemos aplicado un kernel lineal, con un valor de C (un parámetro de regularización) de 1, un máximo de 20 iteraciones y un valor de tolerancia de  $1e-3$ .

El resultado ha sido un 100% de datos de entrenamiento clasificados correctamente y un 67.96% de datos de test clasificados correctamente. Estos resultados se han obtenido en 270.02 segundos.

Después hemos aplicado un kernel gaussiano (en la función *gaussianKernel.m*), con un valor de C de 1 y un valor de sigma de 0.1. Estos cálculos han tardado 50.64 segundos y el resultado ha sido un 100% de datos de entrenamiento clasificados correctamente y un 90.71% de datos de test clasificados correctamente.

El código de la función es el siguiente:

A screenshot of a MATLAB script editor window. The window has a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The script defines a function named 'sim' as 'gaussianKernel(x1, x2, sigma)'. The function body calculates the squared distance between x1 and x2, divides it by twice the square of sigma, and then takes the negative exponential of that result. The function ends with 'endfunction'.

```
function sim = gaussianKernel(x1, x2, sigma)
    sim = exp(-(sum((x1-x2).^2)/(2*(sigma^2))));
endfunction
```

Después hemos iterado con el kernel gaussiano para calcular el mejor valor para sigma y C. Hemos obtenido un valor de C óptimo de 0.01 y un valor óptimo de sigma de 0.1. Estos valores dan como resultado un 99.56% de datos de cross-validation clasificados correctamente y un 90.71% de datos de test clasificados correctamente.

También hemos calculado la precisión y el recall de este algoritmo y ha sido respectivamente 91.55% y 98.82%.

El código de nuestra función (*svm.m*) será el siguiente:

```
function svm()
    fflush(stdout);
    warning('off','all');

    addpath('SVM');
    file = 'mushroomdata';

    printf('Aplicar Support Vector Machines. \n');
    printf('Pulsa una tecla para continuar...');
    % pause();
    printf('\n');

    load(file);
    [datatrain, datacrossvalidation, datatest] = loadandrandomize(file);
    [X, y] = splitdataInvars(datatrain);
    [Xval, yval] = splitdataInvars(datacrossvalidation);
    [Xtest, ytest] = splitdataInvars(datatest);

    printf('Primero entrenamos la Support Vector Machine con un kernel lineal\n');

    tic;
    linearModel = svmTrain(X, y, 1, @linearKernel, 1e-3, 20);
    time = toc;
    printf('Se ha calculado el modelo con un máximo de 20 iteraciones y un valor de C = 1 \n');
    printf('El cálculo ha durado %.2f segundos\n', time);
    printf('Pulsa una tecla para continuar...');
    % pause();
    printf('\n');

    prediction = svmPredict(linearModel, X);
    percentage = sum(y == prediction)/rows(y);
    printf('Teniendo en cuenta que vamos a testear el porcentaje de acierto con los mismos datos que los que hemos usado para entrenar nuestro algoritmo, obtenemos un porcentaje de acierto de %.2f%% \n\n', percentage * 100);

    clear prediction percentage;

    prediction = svmPredict(linearModel, Xtest);
    percentage = sum(ytest == prediction)/rows(ytest);
    printf('Calculando el porcentaje de acierto sobre los datos de test obtenemos un porcentaje de acierto de %.2f%% \n\n', percentage * 100);

    clear prediction percentage;

    printf('Ahora entrenamos la Support Vector Machine con un kernel gaussiano\n');
    tic;
    gaussianModel = svmTrain(X, y, 1, @(x1,x2) gaussianKernel(x1,x2,0.1));
    time = toc;
    printf('El cálculo ha durado %.2f segundos. \n', time);
    printf('Pulsa una tecla para continuar...');
    % pause();
    printf('\n');

    prediction = svmPredict(gaussianModel, X);
    percentage = sum(y == prediction)/rows(y);
    printf('Teniendo en cuenta que vamos a testear el porcentaje de acierto con los mismos datos que los que hemos usado para entrenar nuestro algoritmo, obtenemos un porcentaje de acierto de %.2f%% \n\n', percentage * 100);

    clear prediction percentage;

    prediction = svmPredict(gaussianModel, Xtest);
    percentage = sum(ytest == prediction)/rows(ytest);
    printf('Calculando el porcentaje de acierto sobre los datos de test obtenemos un porcentaje de acierto de %.2f%% \n\n', percentage * 100);

    clear prediction percentage;

    [model, C, sigma, percentagecv] = chooseCandSigma(X, y, Xval, yval);
    prediction = svmPredict(model, Xtest);
    percentage = sum(ytest == prediction)/rows(ytest);
    printf('Los valores para C y para sigma optimos son %.2f y %.2f respectivamente. \nEl porcentaje de acierto con esos valores en los datos de cross-validation es %.2f%% \nEl porcentaje de acierto sobre los datos de test es %.2f%% \n\n', C, sigma, percentagecv * 100, percentage * 100);

    truepos = ytest + prediction == 2;
    if (prediction == 0)
        precision = 1;
    else
        precision = sum(truepos) / sum(prediction);
    end
    recall = sum(truepos) / sum(ytest);

    printf('Este algoritmo tiene una precision de %.2f%% y un recall de %.2f%%. \n', precision * 100, recall * 100);

endfunction
```

## CONCLUSIONES

Tras entrenar la support vector machine podemos obtener una serie de conclusiones.

Comparándolo con la regresión logística y la red neuronal obtenemos muy buenos resultados. El porcentaje de acierto es mucho mayor, con una diferencia de casi un 30% más.

Por tanto, deducimos que en este problema podría ser adecuado aplicar una support vector machine ya que el beneficio obtenido es enorme, aunque eso sí, sacrificando el coste de ejecución que aumenta significativamente.

Los porcentajes de precisión y recall aumentan también excesivamente, lo que supone otro argumento más a favor de aplicar support vector machines en este problema.

El recall es casi ideal, ya que es 98.82% lo que significa que de cada 100 setas venenosas en nuestro subconjunto de test nos está informando de 98 y la precisión (la "seguridad" que tiene al decir que una seta es venenosa) es también bastante alta (91.55%).

## **Tabla comparativa de los 3 algoritmos**

Método	Neuronas en la capa oculta	Regularización	Tiempo	Mínimo valor de la función de coste	Aciertos training set	Aciertos cv set	Aciertos test set	Mejor threshold	Precisión	Recall
Regresión logística sin división de datos	N/A	Lambda=0.01	0.64s	0.085527	98.09%	N/A	N/A	N/A	N/A	N/A
Regresión logística con división de datos	N/A	Lambda=0.01	N/A	0.0056585	N/A	98.05%	66.37%	0.11	88.78%	72.23%
Red neuronal con todos los datos en entrenamiento	3	Lambda=0.01	2.29s	0.003449	100.00%	N/A	N/A	N/A	N/A	N/A
Red neuronal con 80% entrenamiento y 20% test	10	Lambda=0.01	4.06s	0.002366	N/A	100.00%	89.82%	N/A	N/A	N/A
Red neuronal con división de datos	10	Lambda=0.1	N/A	0.010507	N/A	98.05%	66.46%	0.01	88.97%	73.60%
SVM (Kernel lineal)	N/A	C=1 Sigma=N/A	270.02s	N/A	100.00%	N/A	67.96%	N/A	N/A	N/A
SVM (Kernel gaussiano)	N/A	C=1 Sigma=0.1	50.64s	N/A	100.00%	N/A	90.71%	N/A	N/A	N/A
SVM (Kernel gaussiano)	N/A	C=0.01 Sigma=0.1	N/A	N/A	N/A	99.56%	90.71%	N/A	91.55%	98.82%