APRENDIZAJE AUTOMÁTICO

UCM - FDI

Práctica Final

Breast Cancer Wisconsin – Algoritmos de aprendizaje automático

DAVID GÓMEZ BLANCO

Índice

Nuestro Dataset 3
Introducción 3
Los datos 3
Procesado de los datos 4
Los algoritmos 5
Regresión logística 5
Aumento polinomial de características 6
Regularización 8
Redes neuronales 10
Support Vector Machines 15
Comprobar el mejor C y sigma 15
Usando ***libsvm*** 16
Conclusiones 23
Ampliando contenido 23
Dataset sobre el pronóstico 23
Extendiendo con Weka 25
Conclusiones 28
Referencias 29

Nuestro Dataset

En este primer apartado se hará una introducción a nuestro dataset y al porqué sobre la necesidad de aplicar distintos algoritmos de aprendizaje automático para la solución de problemas tales como el diagnóstico y el pronóstico.

Se explicará tanto la historia (brevemente) como el formato de los datos, su dominio y el tratamiento inicial de los datos para empezar a trabajar con los algoritmos de aprendizaje automático. Si bien, la lectura que viene a continuación no es obligada pero si interesante para entender lo hecho en los algoritmos.

Introducción

Nuestro dataset nace de la mano del Dr. Wolberg, un doctor de la universidad de Wisconsin que usaba los datos clínicos de sus pacientes desde 1989 hasta 1991. Los datos que refleja nuestro dataset se corresponden al uso de un método de separación de patrones multicapa para el diagnóstico del cáncer de mama. Gracias a este método, el doctor podía clasificar las diferentes variables del cáncer y aplicarles un valor numérico cuantificable.

Aunque empezó siendo un dataset de 367 instancias, acabó siendo lo que es hoy desde 1991, 699 instancias. Todos los usos anteriores mencionados en la documentación del dataset son anteriores a 1991 y por lo tanto hechos con menos instancias.

Los datos

Como ya se ha dicho antes, el dataset contiene 699 instancias, las cuales poseen 10 atributos más 1 que define la clase.

Número	Atributo	Dominio
1	Sample code number	ID
2	Clump Thickness	1 - 10
3	Uniformity of Cell Size	1 - 10
4	Uniformity of Cell Shape	1 - 10
5	Marginal Adhesion	1 - 10
6	Single Epithelial Cell Size	1 - 10
7	Bare Nuclei	1 - 10
8	Bland Chromatin	1 - 10
9	Normal Nucleoli	1 - 10
10	Mitoses	1 - 10
11	Class	2 = benign 4 = malignant

No es interesante, para nosotros, conocer el significado de cada uno de los atributos, aunque si es recomendable para tener un conocimiento global sobre el objeto de estudio.

Además, cabe destacar que en el dataset hay 16 instancias con atributos perdidos, es decir, ciertos atributos tienen un valor '?' en su valor, denotando que, en el momento de creación de la instancia, no se tenía muy claro su valor.

Por último, hablar sobre la distribución de las clases: el 65.5% son instancias benignas (458), mientras que el 34.5% restante corresponden a instancias malignas (241). Esta distribución es importante tenerla en cuenta, ya que puede ser que los datos no estén bien distribuidos a lo largo del dataset.

Procesado de los datos

Antes de empezar a aplicar nuestro algoritmos de aprendizaje automático hay que tratar los datos, ya que tal y como están no son usables.

Para empezar, salta a la vista que el atributo *Sample Code Number* nos sobra para el correcto funcionamiento de los algoritmos, ya que su valor no guarda relación con el resultado de la clase, así que directamente podemos eliminar ese atributo.

Para eliminarlo, primero tenemos que leer los datos. Como se recordará, hay ciertas instancias que tienen valores '?' en sus atributos. Estos valores serán sustituidos por 0 por comodidad, no son relevantes en el correcto funcionamiento ya que son muy pocos.

```
bcwdata = dlmread( "breast-cancer-wisconsin.data" , "," );
bcwdataC = columns( bcwdata );
bcwdata = bcwdata( : , [2:bcwdataC] );
bcwdataC = columns( bcwdata );
bcwdataR = rows( bcwdata );
bcwdataX = bcwdata( : , [1:bcwdataC-1] );
bcwdataY = bcwdata(: , bcwdataC );
```

Con este código es suficiente para tratar los datos tal y como exponemos arriba. Solo queda el tratamiento de la clase: que venga denotada por 2 y 4… no nos gusta.

Para facilitar la lectura, usaremos un código binario para describir si un cáncer en benigno o maligno (0 y 1 respectivamente). Esto es mucho más fácil de lo que se cree, solo es necesaria una línea de código.

```
bcwdataY = ( bcwdataY == 4 );
```

Fácil ¿verdad? Ahora si que tenemos nuestros datos completamente tratados. Ya podemos pasar a usar los algoritmos de aprendizaje automático que hemos aprendido en la asignatura. Pero antes, vamos a crear algunas variables que nos serán de utilidad durante las pruebas.

```
lambda = [0.001 , 0.003 , 0.01 , 0.03 , 0.1 , 0.3 , 1 , 3 , 10 , 30 , 100 , 300 ];
BESTlambda = 0;                    #Mejor valor de lambda encontrado
BESTprc = 0;                       #Mejor porcentaje de ejemplos bien clasificados
num_et = length(unique(bcwdataY)); #Número de etiquetas de clasificación
```

Ahora si estamos preparados para comenzar a probar diferentes configuraciones con los algoritmos propuestos.

Los algoritmos

Como nuestro problema es un problema de clasificación, podemos usar más de la mitad de los algoritmos aprendidos en clase. Estos algoritmos son:

- Regresión logística
- Redes neuronales
- *Support Vector Machines*

Los iremos viendo por orden, explicando el proceso seguido de ejecución y las motivaciones encontradas para hacer una cosa u otra. Se mostrarán también datos y gráficas que ayudarán a la comprensión de los datos obtenidos.

En principio todos los algoritmos siguen el mismo principio:

- Primero se hará una ejecución sin regularizar con el dataset. Los datos obtenidos se mostrarán así como las gráficas de sesgo y varianza.
- Dependiendo del resultado de los primeros datos y las gráficas, se usarán distintos métodos para solucionar los posibles errores de sesgado o varianza sin reducir, en la medida de lo posible (o mejorando), el resultado inicial.

Regresión logística

Empezamos a usar nuestros algoritmos con el primer algoritmo que vimos de clasificación, ya que nuestro dataset es de ese tipo: **Regresión Logística**.

Primero de todo, haremos una ejecución normal, sin parámetro de regularización o *lambda* igual a 0 y miraremos su curva de aprendizaje para ver si peca de sesgado o de varianza. El dataset se dividirá en 2 porciones: un 70% para ejemplos de entrenamiento y un 30% para ejemplos de validación.

La ejecución resultó en **la gráfica 1** que se expone más abajo y, como se puede observar, tenemos un problema de sesgado. Esto lo sabemos porque la gráfica acaba convergiendo para valores altos del número de ejemplos.

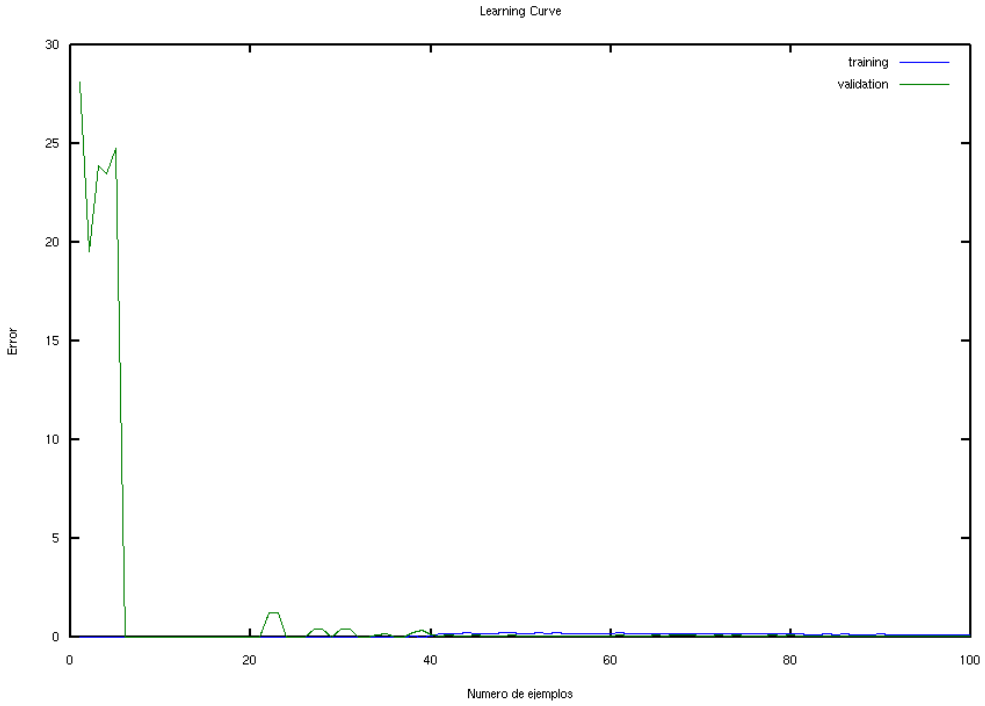


Figura 1: Ejecución normal del algoritmo de regresión logística sin parámetro de regularización.

¿Cómo podemos solucionar este problema? Tenemos dos posibles soluciones a nuestro problema y, por supuesto, veremos todas ellas:

- Aumentar polinómicamente el número de características.
- Aumentamos el parámetro *lambda* o índice de regularización.

Empezaremos con el aumento polinomial.

Aumento polinomial de características

La motivación para usar este método es que incrementando el grado de nuestra hipótesis (aumentando el número de características) conseguimos disminuir el sesgado, resolviendo así nuestro problema.

Para poder realizar la combinatoria que supone el incremento polinomial de atributos, se ha uso la biblioteca ***specfun***, más concretamente, la función ***multinom***, que lo que hace es generar, con unos valores **X** y un **grado** pasados por referencia, la combinatoria de **X** con grado **grado.**

```
function [y, alpha] = multinom(x,n,sortmethod)
   [nT, m]  = size(x);
   if nargin > 2  alpha = multinom_exp(m,n,sortmethod);
   else  alpha = multinom_exp(m,n);
   end
   na = size(alpha,1);
   y = reshape( prod(repmat(x,na,1).^kron(alpha,ones(nT,1)),2) ,nT,na);
end
```

Para más información, en la parte de **Referencias** se incluirá un link a la biblioteca.

Ya podemos empezar la prueba. Se prueban con varios grados pero, a partir de 5, el cálculo se hace insoportable para el ordenador.

```
degrees=[2, 3, 4, 5];
for i=1:length(degrees)
        Xaux = multinom(bcwdataX, degrees(i));
        [Xfin, mu, sigma] =featureNormalize(Xaux);
        [Xtrain,Xval,Xtest,ytrain,yval,ytest] = datasetPart(Xfin, bcwdataY, 0.7, 0.3, 0.0);

        Xtrain = bsxfun(@minus, Xtrain, mu);
        Xtrain = bsxfun(@rdivide, Xtrain, sigma);
        Xval = bsxfun(@minus, Xval, mu);
        Xval = bsxfun(@rdivide, Xval, sigma);

        learningCurve(Xtrain, ytrain, 0, Xval, yval, 100);
end
```

Donde la función ***datasetPart*** es de creación propia y ***learningCurve*** ha sido modificada de la original presentada en clase.

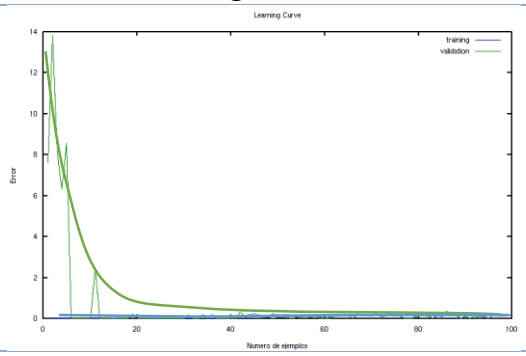
La primera función divide el dataset en 3 porciones con los porcentajes enviados por parámetro. Así, la primera, con 0`7, serían los **Xtrain** y el 0'3 sería el **Xval**.

```
function [Xtrain, Xval, Xtest, ytrain, yval, ytest] = datasetPart(X, y, trainPrc, valPrc,
testPrc)
                XtrainL = (trainPrc)*rows(X);
                XvalL = (valPrc)*rows(X);
                XtestL = (testPrc)*rows(X);
                if(ceil(XtrainL)-XtrainL < 0.5) XtrainL = ceil(XtrainL);
                else XtrainL = floor(XtrainL);
                endif
                ###LO MISMO PARA Xval y Xtest ###
                endif
                Xtrain = X(1:XtrainL,:);
                Xval = X(XtrainL+1:(XtrainL+XvalL),:);
                Xtest = X((XtrainL+1+XvalL):(XtrainL+XvalL)+XtestL,:);
                ytrain = y(1:XtrainL,:);
                yval = y(XtrainL+1:(XtrainL+XvalL),:);
                ytest = y((XtrainL+1+XvalL):(XtrainL+XvalL)+XtestL,:);
        endif
endfunction
```

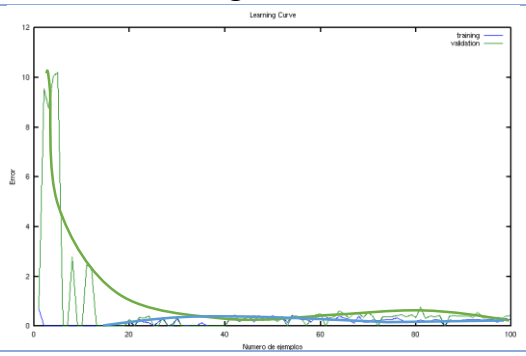
El cambio principal que se ha realizado en ***learningCurve*** es la inclusión de la función de regresión logística, por eso, considero innecesario colocar aquí el código. En base, es igual que el presentado en las memorias de las prácticas anteriores.

Tras la ejecución del bucle para varios grados los resultados obtenidos fueron:

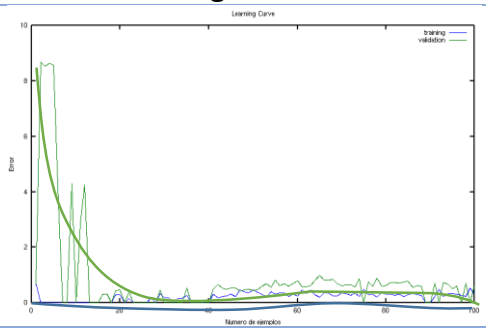
Degree = 2



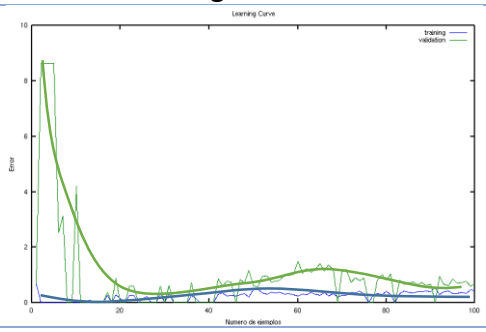
Degree = 3



Degree = 4



Degree = 5



Es notable la mejora en el sesgado conforme aumentamos el número de características. Para un grado 5, que es el máximo que hemos probado, ya se ve la diferencia comparándolo con 2.

Conclusión

No merece la pena la mejora obtenida con respecto al tiempo de ejecución que conlleva. Creo que es mejor mantener un valor de grado 1 y probar otros métodos. Aún nos queda el índice de regularización.

En el peor de los casos, si la regularización no da los frutos que queremos, siempre podemos mezclar el aumento polinomial con la regularización.

Regularización

La técnica llamada regularización nos permite reducir el sobreajuste. Consiste en reducir la importancia de los parámetros **Theta** que aparecen en la función de coste.

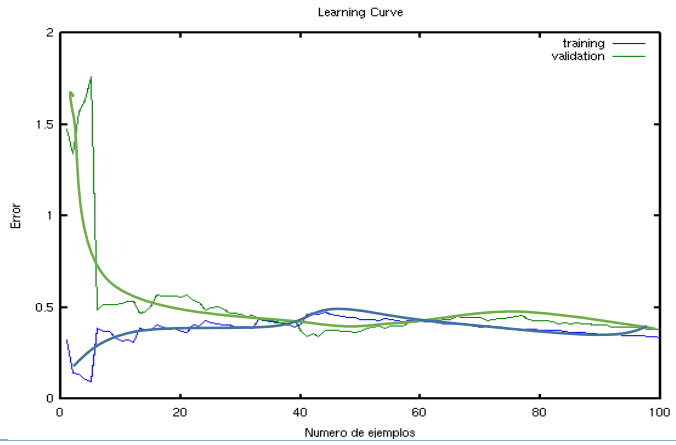
Sabiendo esto, aumentaremos nuestra **Theta** hasta un valor que nos deje de dar las prestaciones que nos venía dando la regresión logística, con un **100% de los datos bien clasificados.**

```
parts=[0.6,0.2,0.2];

for j=1:rows(parts)
        for i=1:length(lambda)
                [Xtrain,Xval,Xtest,ytrain,yval,ytest] = datasetPart(bcwdataX, bcwdataY,
        parts(j,1), parts(j,2), parts(j,3));
                learningCurve(Xtrain, ytrain, lambda(i), Xval, yval, 100);
                [theta, cost] = logReg(Xtest, ytest, lambda(i));
                prc = check(Xtest, ytest, theta);
                if BESTprc <= prc
                        BESTprc = prc;
                        BESTlambda = lambda(i);
                endif
        end
end
```

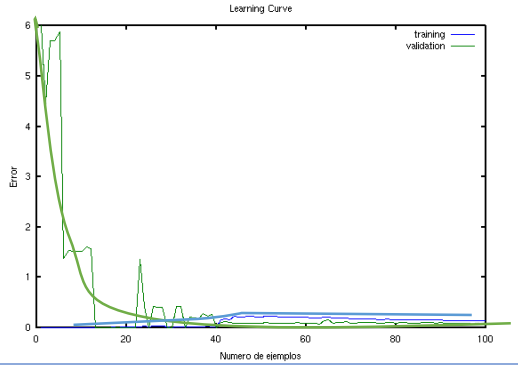
Como se puede ver arriba, se probará con la distribución 0'6-0'2-0'2 Donde el último valor se usará para el **Xtrain**, aquellos ejemplos que se usarán para ver como de buena es nuestra hipótesis.

Al final de la ejecución, el resultado fue:

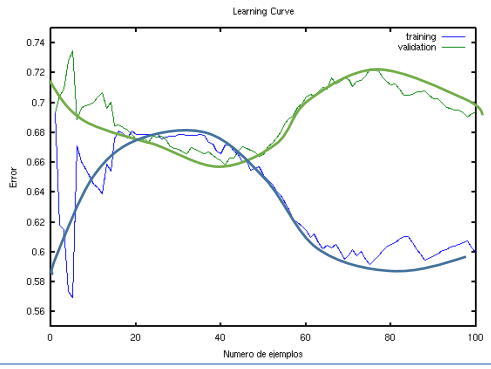
Lambda = 3



Lambda = 0.001



Lambda = 300



Creo que sobran las palabras ¿Verdad? La mejora obtenida con *lambda 3* es infinitamente mejor que la obtenida con el incremento polinomial. En la figura de arriba, se tiene la comparativa entre el menor de las lambdas y el mayor de los mismos probados.

Nos quedamos con *lambda 3* porque es la mayor de las lambdas que aún nos ofrece un **100% de resultados** en la clasificación.

Conclusión

No nos hace falta para nada el incremento polinomial. Si queremos más características usaremos una Red Neuronal (En el apartado siguiente hablaremos de ello).

Siempre que sea posible, agregar un parámetro de regularización a nuestro problema hará que podamos manipular este a nuestro antojo, y tan solo modificando un valor.

Redes neuronales

Lo primero que debemos hacer para toda red neuronal (no solamente la de la práctica) es buscar la arquitectura idónea para nuestra red. Esta arquitectura será la que de mejores resultados, obviamente.

Toda red neuronal se ve definida por tres factores:

- Número de nodos de entrada: Corresponde al número de características que tiene nuestro dataset o conjunto de ejemplos de entrenamiento. En nuestro caso es fácil descubrirlo, solo hace falta mirar el número de columnas de **X**
- Número de nodos de salida: Corresponde al número de clases o etiquetas que tiene nuestro dataset. En nuestro caso, las dos clases que tenemos son benigno y maligno, por lo tanto, el número de nodos de salida será 2.

  ```
  entradas = bcwdataC;
  salidas = num_et;
  ```

  Es aquí donde se hace uso de las variables **bcwdataC** y **num_et**, creadas al principio de nuestro ***script*** con la excusa de que nos serían útiles durante la prueba.
- Número de nodos/capas ocultos: Estos nodos son los encargados del tratamiento de los datos haciendo uso de nuestras hipótesis **Theta1** y **Theta2**, que transforman los datos de entrada a ocultos y de ocultos a salida respectivamente.
  Aquí es donde radica toda la elección de nuestra arquitectura. Por regla general, lo mejor es añadir muchas capas/nodos y hacer varias

iteraciones en la retropropagación (explicada en la memoria de Redes Neuronales). Por eso, en nuestra ejecución, vamos a probar varias iteraciones y varias capas/nodos ocultos.

La ejecución de las redes neuronales tardaba muchísimo tiempo, además, si encadenabas dos bucles para ver la mejor relación entre parámetros, muchísimo más.

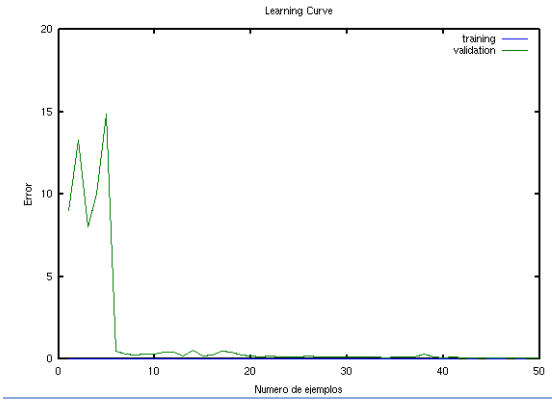
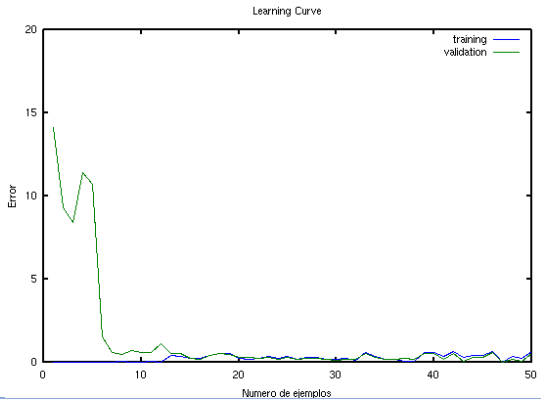
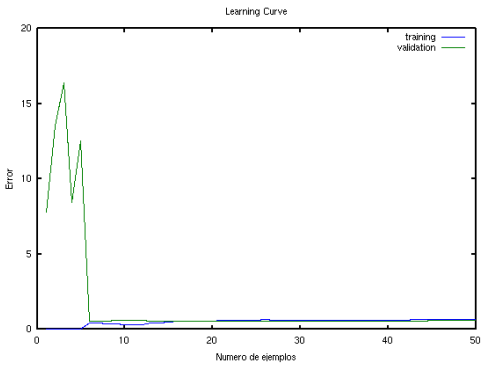
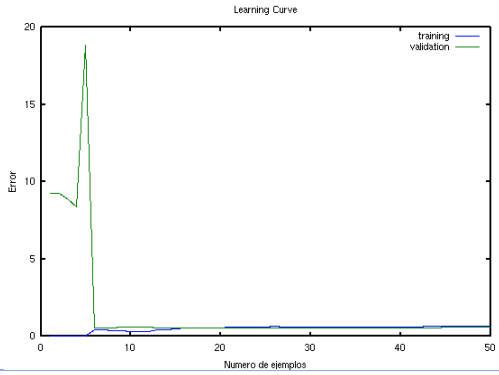
Es por eso que, para no tener que estar pendiente de cuando salían los resultados para lanzar a ejecución otros, hice todo de golpe.

```
for i = 1:length(lambda)
        for j = 1:length(oc)
                for k = 1:length(it)
                        theta1 = pesosAleatorios(entradas,oc(j));
                        theta2 = pesosAleatorios(oc(j),salidas);

                        [Xtrain,Xval,Xtest,ytrain,yval,ytest] = datasetPart(bcwdataX,
bcwdataY, 0.6, 0.2, 0.2);

                        options = optimset('MaxIter', it(k));
                        coste = @(p) costeRN(p, entradas, oc(j), salidas, Xtrain, ytrain,
lambda(i));
                        matriz_enrollada = rolling(theta1,theta2);
                        gradiente = fmincg(coste, matriz_enrollada, options)
                        grad1 = reshape(gradiente(1:oc(j) * (entradas + 1)), …
                          oc(j), (entradas + 1));
                        grad2 = reshape(gradiente((1 + (oc(j) * (entradas + 1))):end), ...
                          salidas, (oc(j) + 1));

                        prc = checkNeural(Xtest, ytest, grad1, grad2, entradas, oc(j),
salidas) ;
                        printf("For(%i) %i hiddens and lambda %f: %f\n", it(k),  oc(j),
lambda(i), prc);
                        fflush(stdout);

                        if BESTprc <= prc
                                BESTprc = prc;
                                BESTlambda = lambda(i);
                                BESToc = oc(j);
                                BESTit = it(k);
                        endif

                        learningCurveRN(Xtrain, ytrain, lambda(i), Xval, yval, it(k), entradas,
oc(j), salidas);
```

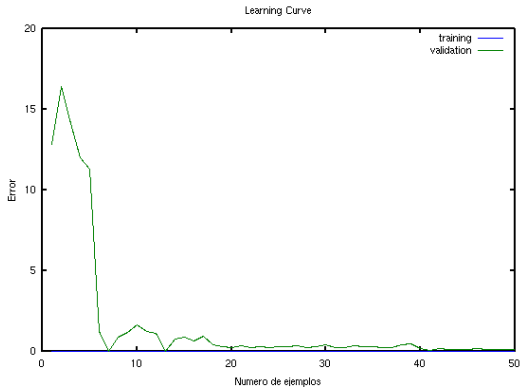
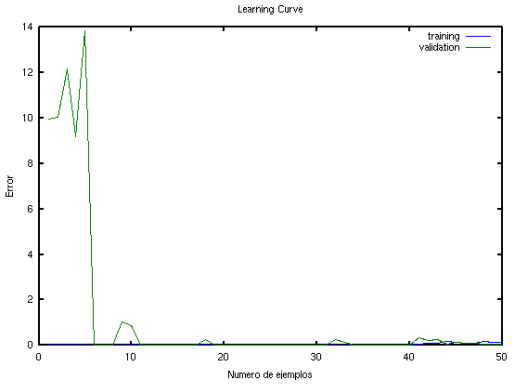
Aquí vemos un triple bucle que calcula a la vez cual es el mejor número de capas ocultas para el mejor número de iteraciones junto con el mejor lambda. Todo de golpe.

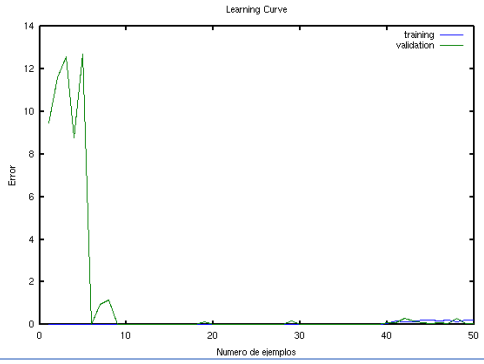
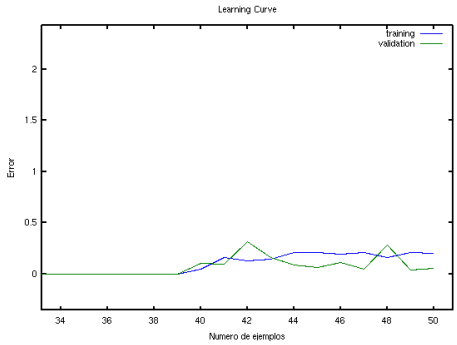
Como ya habíamos hecho una primera ejecución en regresión logística no nos hacía falta hacerlo de nuevo para saber que teníamos un problema de sesgado, así que directamente probamos a aumentar la *lambda* que, como recordareis, fue lo que mejor resultados nos dio.

Decidimos en cada iteración cual es el mejor de cada uno observando el porcentaje de elementos bien clasificados en todo momento. A continuación, habrá una tabla con los datos obtenidos más significativos:

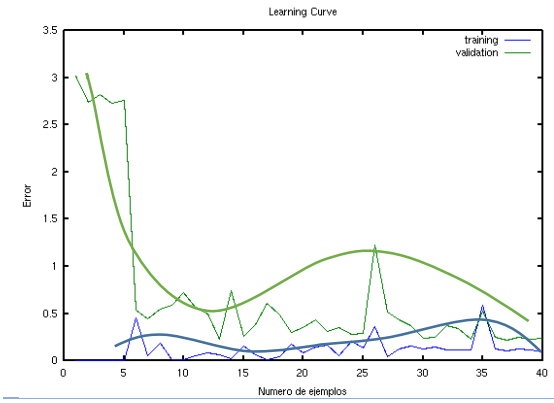
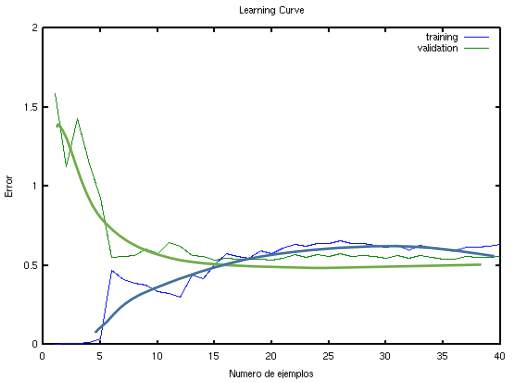
Iteraciones = 10	**Iteraciones = 40**
	
Iteraciones = 160	**Iteraciones = 320**
	

En las gráficas de arriba vemos cómo cambia el aprendizaje con la misma lambda, distinto número de iteraciones y mismo número de capas ocultas. Aunque parezca que el número de iteraciones con las gráficas no es determinante en absoluto, en el apartado de conclusiones expongo el porqué de este problema y porque no pude solucionarlo aunque lo intenté.

Capas ocultas = 200	Capas ocultas = 2000
	

Capas ocultas = 6000	Ampliación 6000
	

En las gráficas de arriba vemos cómo cambia el aprendizaje con la misma lambda, mismo número de iteraciones y distinto número de capas ocultas. Se puede observar, gracias a la ampliación de 6000 como empieza a notarse que los ejemplos de entrenamiento empiezan a superar a los de validación. Aun así, no es el resultado que esperaba. Lo mismo que antes, la explicación, en conclusiones.

Lambda = 0.001	Lambda = 300
	

En las gráficas de arriba vemos cómo cambia el aprendizaje con distinta lambda, mismo número de iteraciones y mismo número de capas ocultas. Se han mostrado los resultados del lambda más pequeño probado y del más grande. Ahora si se nota la diferencia. Vemos como, de nuevo aumentar el índice de regularización mejora notablemente el rendimiento de nuestro algoritmo.

Aún con todo lo mostrado, seguimos con el mismo problema: la red no clasifica todo lo bien que pudiera nuestro dataset: El máximo valor que se ha alcanzado ha sido 75% y ha sido alcanzado por cualquier Nº de iteraciones, 160 o 320 capas ocultas y cualquier lambda. El peor valor obtenido fue 61,428%, para el Nº de iteraciones 10, lambda 0.001 y capas ocultas 200.

Conclusiones

Navegando un poco por internet para saber porque clasificaba "tan mal" (viendo los resultados de la regresión logística) mi dataset las redes neuronales descubrí que las redes neuronales son muy variantes dependiendo de la naturaleza de los dataset. Por ejemplo, el dominio de los atributos, cuantos atributos tiene…

Si estos valores eran muy pequeños o poco cambiantes, entonces harán falta muchas iteraciones y muchos nodos ocultos para poder obtener unos resultados esperados. Cuando se dice "pequeños o poco cambiantes" se refiere al número de atributos y a su dominio respectivamente.

Las redes neuronales funcionan tan bien en reconocimiento de imágenes por ese motivo: la cantidad de datos que puede albergar un pixel (todo su dominio flotante) comparado con los 9 valores que puede tomar mi dataset se queda corto. Lo mismo pasa con el número de atributos.

Probé enseguida a realizar el cálculo con muchas más iteraciones (+1000) y con un número de capas ocultas ingente (+5000), pero no pude obtener ningún valor válido pues siempre aparecía lo mismo:

```
Iteration    1550 | Cost: Nan995e-06   1 | Cost: 1.9264e-01
Iteration    1550 | Cost: Nan033e-07   1 | Cost: 5.7408e-01
Iteration    1550 | Cost: Nan010e-02   1 | Cost: 5.6602e-01
Iteration    1550 | Cost: Nan986e-02   1 | Cost: 6.5620e-01
Iteration    1550 | Cost: Nan073e-02   1 | Cost: 5.8547e-01
Iteration    1550 | Cost: Nan380e-02   1 | Cost: 5.3075e-01
Iteration    1550 | Cost: Nan516e-02   1 | Cost: 6.4894e-01
Iteration    1550 | Cost: Nan858e-03   1 | Cost: 7.4242e-01
```

Esos valores con **Nan** no resultaban válidos de cara al programa, por eso, no pude obtener ningún dato. Tuve que dejar de aumentar las iteraciones por este problema que intenté buscar solución pero no encontraba nada.

Por todo lo contado anteriormente creo que podemos sacar dos conclusiones bastante satisfactorias.

- Para datasets con pocos atributos o poco dominio de los mismos, lo mejor es aplicar una regresión logística. El tiempo computacional es mucho mejor y, aunque con las redes neuronales potencialmente puedes obtener mejores resultados, su gasto computacional es enorme.

- Para dataset con muchos atributos y mucho dominio de los mismos, lo mejor es una red neuronal. Gracias a las propiedades de la misma puedes "jugar con su arquitectura" y probar distintas redes para ver cuál es la que mejor se adapta, no solo a tu dataset actual, si no a futuras ampliaciones del mismo.

Support Vector Machines

Debido a los problemas que hubo con el algoritmo suministrado en clase, yo y otros compañeros tuvimos que descargarnos de internet la librería ***libsvm*** para poder realizar la prueba de las *Support Vector Machine.*

Aprovechando que tenía que usar una biblioteca nueva con varias opciones no vistas en clase, aproveché para indagar más sobre estas y usarlas en mi algoritmo. Ahora pasaré a hablar de ellas.

Comprobar el mejor C y sigma

Una cosa que no sabemos si podemos hacerlo con el código suministrado es la elección del mejor parámetro **C** y el mejor parámetro **sigma** para el kernel gaussiano, así que vamos a comprobarlo y, dependiendo del valor que nos salga, sabremos si se puede hacer o no.

Para ello, hay que entrenar primero con ***svmtrain*** y comprobar el modelo generado con ***svmpredict***.

```
CJ = [0.001,0.003,0.01,0.03,0.1,0.3,1,3,10,30];
total = rows(bcwdataX);
BEST_C = 0.001;
prctj = 0;
for i = 1:length(CJ)
        printf("Calculo del modelo de SVM para C = %f y sigma = %f\n", CJ(i), CJ(j));
        models(i,j) = svmTrain(bcwdataX, bcwdataY, CJ(i), @linearKernel, 1e-3,
20);#@(x1,x2) gaussianKernel(x1,x2,CJ(j)));
        printf("Validando resultado...\n");
        tmp = sum(svmPredict(models(i,j), bcwdataX) == bcwdataY)/total;
        if prctj < tmp
                prctj = tmp
                BEST_C = CJ(i);
        end;
end;
```

El resultado de ejecutar este código resulta un tanto raro. El número de ejemplos bien clasificados es el mismo sea cual sea el valor de **C (34.47%)**. Creo que, efectivamente, esto es debido al código usado. Este estaba pensado para ser usado en un dataset compuesto de 0 y 1 lógicos organizados en un vector.

El programa al ejecutarse ya nos estaba avisando de que algo no iba bien (están marcados en rojo los mensajes.

Calculo del modelo de SVM para C = 0.001000 y sigma = 0.001000
warning: value not equal to 1 or 0 converted to logical 1

Training ..... Done!

Validando resultado...
prctj = 0.34478
Actualizando el mejor valor de C = 0.001000 y sigma = 0.001000...
Calculo del modelo de SVM para C = 0.001000 y sigma = 0.003000
warning: value not equal to 1 or 0 converted to logical 1

Training ..... Done!

Validando resultado...
Calculo del modelo de SVM para C = 0.001000 y sigma = 0.010000
warning: value not equal to 1 or 0 converted to logical 1

No nos queda otra que usar la biblioteca ***libsvm***.

Usando *libsvm*

En **Referencias** incluiré un enlace a la página donde puede ser descargado la librería. Esta, se compone de varias funciones pensadas para varios lenguajes de programación como Java y MATLAB. Para nosotros, el único interesante es el de MATLAB, que es el que usaremos.

Indagando un poco descubro una herramienta bastante particular: **SVM-TOY**, un interfaz gráfico de plotting para dataset con algoritmos de svm. Me pareció una herramienta curiosa ya que, con unos pocos comandos y un data set (convertido el formato eso sí), puedes comparar distintos kernels, valores de C, sigmas… todo lo relacionado con las SVM's.

Lo primero, para empezar a usarlo, es modificar el archivo de texto de nuestro dataset. Para ello, se especificó una función que convierte mi dataset en el buscado por el programa.

```
fid = fopen('data.txt', 'w');
for i=1:rows(bcwdataX)
        if(bcwdataY(i) == 0)
                fprintf(fid, '1 ');
        else
                fprintf(fid, '2 ');
        end
        for j=1:columns(bcwdataX)
                fprintf(fid, '%i:%f ', j, bcwdataX(i,j)/10);
        end
        fprintf(fid, '\n');
end
fclose(fid);
```

Este programa transforma nuestra matriz de datos en un archivo de texto legible por el programa:

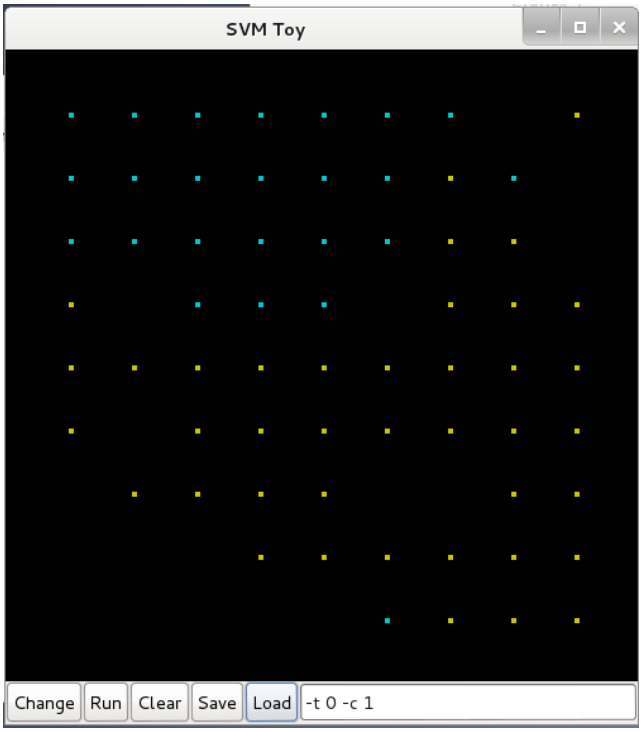
```
1000025,5,1,1,1,2,1,3,1,1,2
1002945,5,4,4,5,7,10,3,2,1,2
1015425,3,1,1,1,2,2,3,1,1,2
1016277,6,8,8,1,3,4,3,7,1,2
1017023,4,1,1,3,2,1,3,1,1,2
1017122,8,10,10,8,7,10,9,7,1,4
1018099,1,1,1,1,2,10,3,1,1,2
1018561,2,1,2,1,2,1,3,1,1,2
1033078,2,1,1,1,2,1,1,1,5,2
1033078,4,2,1,1,2,1,2,1,1,2
1035283,1,1,1,1,1,1,3,1,1,2
1036172,2,1,1,1,2,1,2,1,1,2
1041801,5,3,3,3,2,3,4,4,1,4
```

```
1 1:0,500000 2:0,100000 3:0,100000
4:0,100000 5:0,200000 6:0,100000
7:0,300000 8:0,100000 9:0,100000
1 1:0,500000 2:0,400000 3:0,400000
4:0,500000 5:0,700000 6:1,000000
7:0,300000 8:0,200000 9:0,100000
1 1:0,300000 2:0,100000 3:0,100000
4:0,100000 5:0,200000 6:0,200000
7:0,300000 8:0,100000 9:0,100000
1 1:0,600000 2:0,800000 3:0,800000
4:0,100000 5:0,300000 6:0,400000
7:0,300000 8:0,700000 9:0,100000
1 1:0,400000 2:0,100000 3:0,100000
```

El segundo archivo tiene el siguiente formato por fila:

<Nº de clase> <Ndº de atributo>:<valor atributo>

Con el formateo hecho, lanzamos a ejecución el nuevo archivo para ver cómo se comporta el programa.

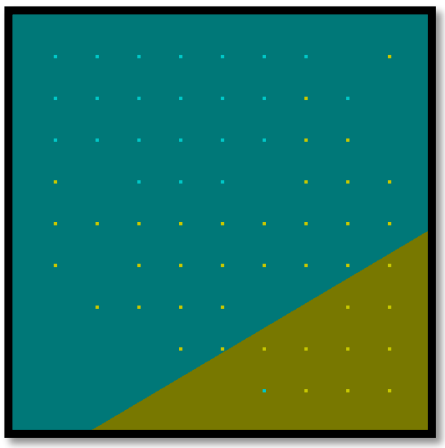
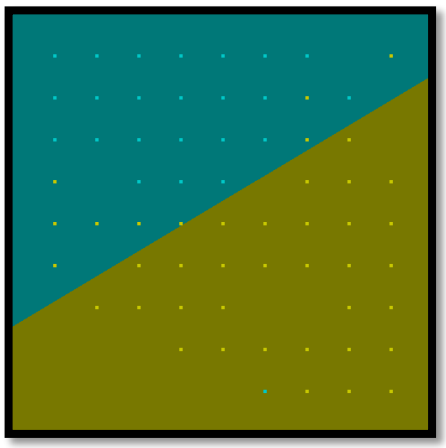


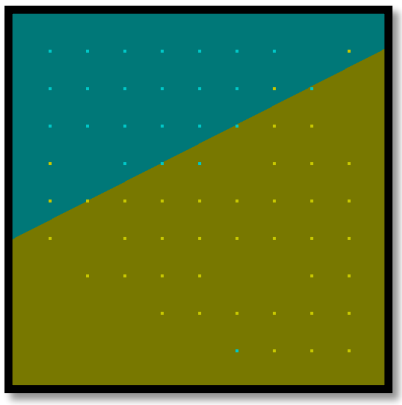
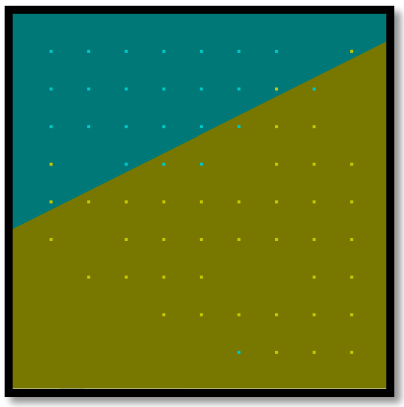
El programa ejecuta por debajo la función ***svmtrain*** para generar el modelo igual a como lo hacíamos nosotros en la Práctica 6. El cuadro de texto que hay abajo sirve para introducir los comandos que se ejecutarán cuando clickemos en '*run*'.

Hay muchos comandos, pero solo usaremos unos cuantos. Los comandos que usaremos y su significado:

- **-t**: Establece que kernel va a ser usado:
  - 0 = Kernel lineal
  - 1 = Kernel Gaussiano
  - 2 = Radial basis function
  - 3 = Sigmoide
- **-c**: Establece el valor de C que quieres usar

Vamos a comprobar la diferencia entre usar una C u otra en kernel lineal.

C = 0,01	**C = 0,1**
	

C = 1	C = 10
	

Es bastante visible como afecta el parámetro **C** a la correcta clasificación de los datos. Para saber el número de ejemplos bien clasificados podemos usar nosotros en un programa de usuario tanto la función ***svmtrain*** como ***svmpredict***.

```
model = svmtrain(double(bcwdataX), double(bcwdataY), '-t 0 -c 1 -q');
tmp = svmpredict(zeros(rows(bcwdataY),1), double(bcwdataY), model,'-q ')
```

Con esto, el resultado es que el porcentaje de aciertos es de **84.37%** para C = 1 y de **87.5%** para C = 10.

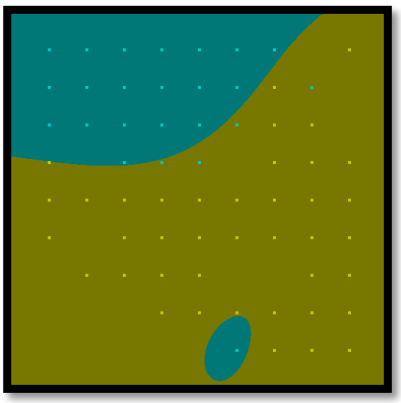
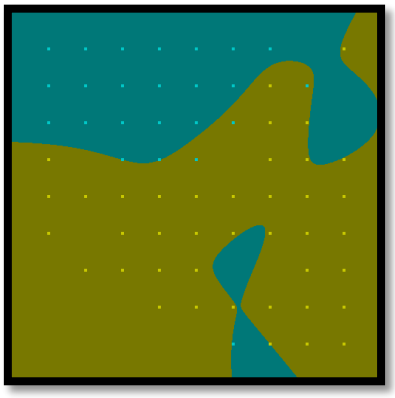
Igual que hemos hecho con el kernel lineal, lo haremos con todos los kernels:

- **Kernel polinomial: (sigma*u'*v + coef0)^degree**

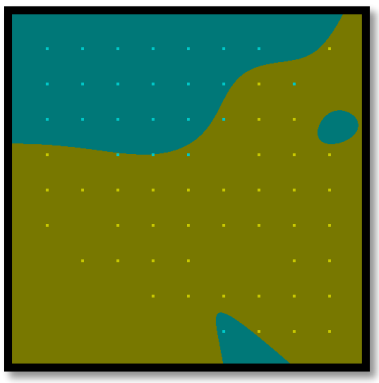
Para este kernel haremos uso de nuevos comandos para especificar el grado polinomial del entrenamiento. Como es habitual pensar, un kernel polinomial se ajustará mucho mejor a nuestros ejemplos de entrenamiento ya que, como se muestra arriba, un kernel lineal no se ajusta perfectamente.

  - **–d**: Establece el grado del polinomio a usar (3 por defecto)
  - **–g:** Establece el valor de sigma.
  - **–r:** Establece el valor de coef.

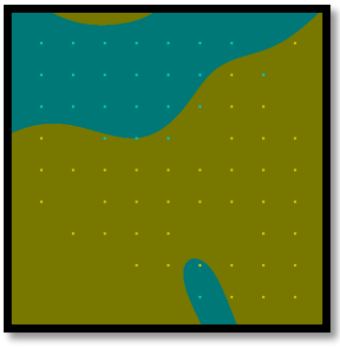
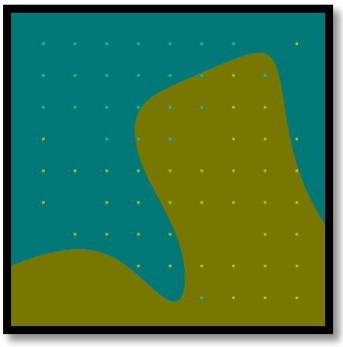
Estableceremos, ahora, las diferencias notables cambiando cada uno de los parámetros.

D = 12	D = 24
	

Las imágenes de aquí son fruto de simplemente modificar el grado. Los demás valores se han mantenido en 1. Se puede observar cómo, al aumentar por 2 el grado, se sobreajusta el modelo. Hay que tener en cuenta que, aunque no se muestra en la figura, el programa tardó casi el doble en generar el modelo de grado 24.

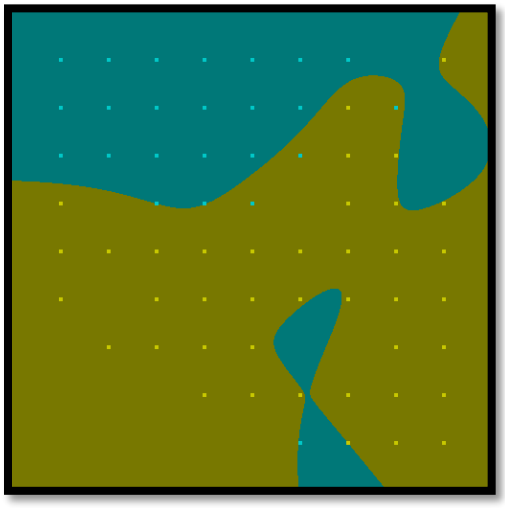
G = 2	G = 10
	

Las imágenes de aquí son fruto de simplemente modificar el valor de sigma. Para d = 12 en los dos casos, se puede ver como el parámetro **sigma** puede modificar y mucho el resultado de la clasificación. Para un sigma de 2 (tan solo 2 veces más que antes) se puede observar como aumenta el sobre ajuste. Lo que ocurre cuando sigma = 10 simplemente es increíble.

R = 2	R = 10
	

Las imágenes de aquí con fruto de simplemente modificar el valor de coef. La verdad, es que no sé qué considerar de estos resultados, parece como si al aumentar el coeficiente, perdiese ajuste y, además, tardase mucho más en hacerlo.

Sabiendo como repercuten las distintas variables a nuestro ajuste, podemos sacar una combinación de las mismas que acierte casi el 100% de la clasificación, modificando también el parámetro C:

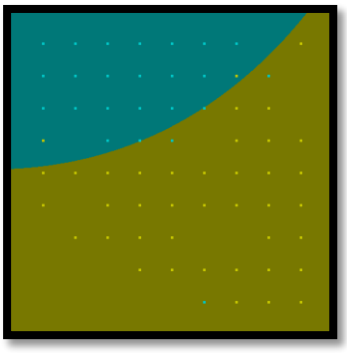
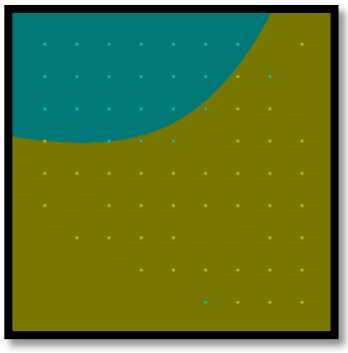


Haciendo uso del código antes mostrado, nos sale una clasificación del **95.34%** Para d = 24, r = 1, g = 1 y C = 10.

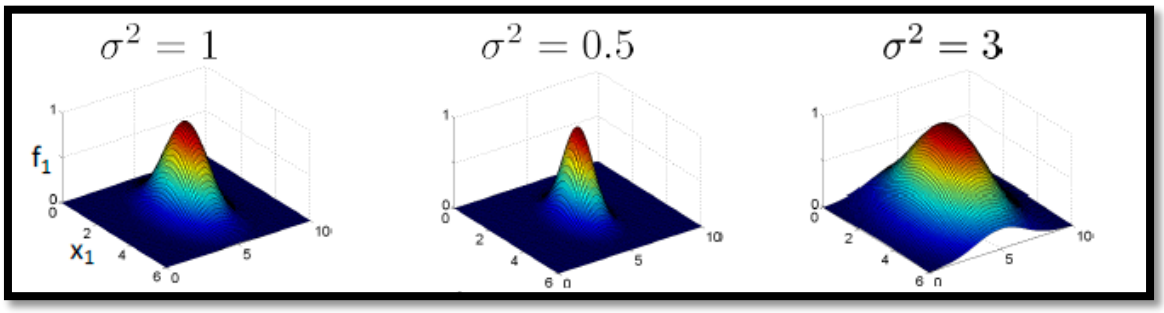
- **Kernel de distancias: exp(-sigma*|u-v|^2)**

Este kernel me recuerda al clustering. Generamos tantos puntos centrales como clases tenga nuestro algoritmo y, dependiendo de la cercanía de los puntos (denotada por la expresión de arriba) los clasifica como de una clase o de otra.

En este apartado solo haremos uso de sigma y de C, por lo que no saldrán tantas gráficas como antes.

G = 1	G = 10
	

Cuanto más alto es el parámetro sigma, menos permisivo es y necesitan los ejemplos estar más cerca del punto central, esto se puede observar muy bien gracias a las funciones de coste.

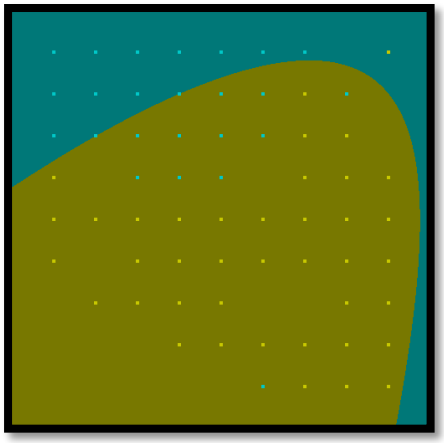


En este caso, el número de ejemplos bien clasificados es **92.18%**, un resultado muy bueno, teniendo en cuenta que no tarda ni por asomo lo mismo que el polinomial y casi consigue los mismos resultados.

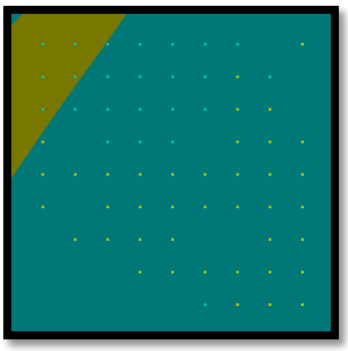
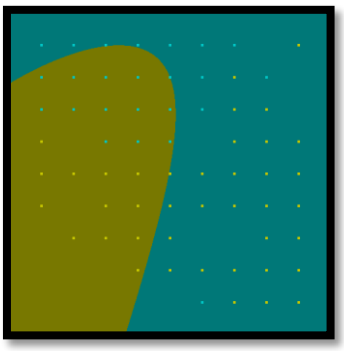
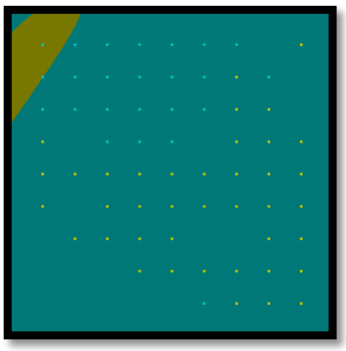
- **Kernel sigmoide: tanh(gamma*u'*v + coef0)**

Curioso lo de este kernel. Ya sabemos cómo funciona la función sigmoide, la hemos visto anteriormente. Para clasificar, toma la imagen de una hipérbole.

Probemos a ejecutarla con todos los parámetros a 1.



No parece que el resultado de esta función nos dé muy buenos resultados o, al menos, muy dispares a los ya obtenidos. Hagamos algunas ejecuciones de prueba para comprobar que esto es cierto.

C = 10 | G = 10 | R = 1	C = 10 | G = 2 | R = 2	C = 1 | G = 10 | R = 2
		

Como decíamos, los resultados no parecen ser buenos. Por eso, obviaremos este kernel que, además, no entraba dentro del temario de la asignatura.

Conclusiones

Primero, una conclusión acerca de SVM: En relación con tiempo de cálculo y ejemplos bien clasificados, SVM se queda con el segundo puesto, por debajo de regresión logística. Pero, si tuviese que elegir un algoritmo, me quedaría con SVM. La cantidad de opciones que te permite es infinitamente mayor que regresión logística. Además, al hacer uso de modelos, no hace falta que relaciones dos atributos como en regresión logística para mostrar como clasifica el algoritmo.

Además, y a lo mejor la razón por la que más me ha gustado, ha sido por el uso de la herramienta SVM-TOY y los algoritmos de la biblioteca **libsvm**. No solo puedes usar más kernels de los aprendidos en la asignatura, si no que puedes jugar literalmente con los parámetros gracias a la herramienta mencionada.

Ampliando contenido

Lo que viene a continuación no es más que una ampliación de los conocimientos aprendidos durante la asignatura. En el primer apartado, se intentará resolver, usando un dataset distinto, si una persona puede volver a tener cáncer de mama y, en caso de que así sea, en cuanto tiempo.

El segundo apartado, haremos uso de la herramienta Weka introducida en la parte de Big Data para comparar nuestros algoritmos con los resultados de Weka. De esta manera, sabremos si lo hemos hecho bien o podemos hacer algo más para mejorar.

Dataset sobre el pronóstico

Este dataset es distinto al anterior: No tiene los mismos atributos ni el mismo número de ejemplos. Nos servirá para pronosticar el cáncer de mama.

Con una simple modificación del código, estamos listos para empezar con el pronóstico:

```
bcwdata = dlmread("wpbc.data",",");
bcwdataC = columns(bcwdata);
bcwdata = bcwdata(:,[2:bcwdataC]);
bcwdataC = columns(bcwdata);
bcwdataR = rows(bcwdata);
bcwdataX = bcwdata(:,[2:bcwdataC]);
bcwdataY = bcwdata(:,1);

lambda = [0.0001,0.001,0.003,0.01,0.03,0.1,0.3,1,3,10,30,100,300];
BESTlambda =
BESTprc = 0;
for i=1:length(lambda)
        [theta, cost] = logReg(bcwdataX, bcwdataY, lambda(i));
        prc = check(bcwdataX, bcwdataY, theta);
        if BESTprc <= prc
                BESTprc = prc;
                BESTlambda = lambda(i);
        endif
end
```

Como resultado de esta ejecución, nos sale que el mejor lambda es **0.0003**, que clasifica los datos con un **88'38%** de seguridad.

Con esto obtenido, podemos pasar a generar un algoritmo que nos diga si un dato es recurrente o no recurrente, esto es, si puede volver a padecer cáncer o no, y, en caso de ser recurrente, en cuanto tiempo podría mostrar síntomas.

```
[theta, cost] = logReg(Xtest, ytest, BESTlambda);
for i=1:rows(Xtest)
        equis = Xtest(i,:);
        r = sigmoide(theta, [1,equis]);
        if r < 0.5
                printf("You're not recurrent!\n");
        else
                printf("You're recurrent...");
                tetaOptima = pinv(transpose(Xtest)* Xtest)*transpose(Xtest)*months;
                tiempo = (hipotesis(tetaOptima, equis));
                printf("Just in %f months\n", tiempo);
        end
end
```

Usamos la **regresión lineal** para averiguar el valor en meses en caso de ser recurrente y comprobamos si es recurrente o no recurrente con regresión logística.

En la práctica 1 de la asignatura, ya se habla de la regresión lineal en múltiples variables, que es la que hemos usado. Así, con una mezcla de conocimientos entre logística y lineal, se puede crear un programa que puede salvar vidas.

Para hacer una prueba, se han usado los ejemplos de entrenamiento 1, 32, 33, 34, 45, 43, 46, 89, 12, 78, 123, 134, 133 y 156:

```
We use lambda = 0.000300 as the best lambda.

You're not recurrent!
You're not recurrent!
You're not recurrent!
You're not recurrent!
You're not recurrent!
You're recurrent...Just in 34.000000 months
You're not recurrent!
You're recurrent...Just in 17.000000 months
You're not recurrent!
You're not recurrent!
You're not recurrent!
You're not recurrent!
You're not recurrent!
You're recurrent...Just in 16.000000 months
Press intro to continue...
```

El valor ha sido acertado para todos los valores excepto para el último, para el cual, el valor se ha excedido 2 meses.

Así, combinando **Regresión lineal** y **Regresión logística** podemos crear un programa que, no solo descubra si una persona puede volver a tener cáncer, si no en cuanto tiempo puede tenerlo.

Extendiendo con Weka

Como se habló en clase, en la parte de Big Data, Weka es una herramienta que, entre otras muchas cosas, te permite cargar un dataset y aplicar distintos algoritmos al mismo, entre ellos, los usados en la asignatura.

Por ello, pensé ¿Por qué no usar la herramienta para ver como de bien he implementado los algoritmos? Así, podría ver si he cometido algún fallo o si puedo hacer más para mejorarlos. Solo se harán pruebas con redes neuronales y con regresión logística.

Empezaremos modificando nuestro dataset original para que Weka pueda entenderlo, al igual que hicimos con SVM-TOY. No voy a explicar el formato de Weka, me limitaré a colocar aquí el resultado del formateo.

```
@relation wisconsin_original
@attribute 'Sample code number' real
@attribute 'Clump Thickness' real
@attribute 'Uniformity of Cell Size' real
@attribute 'Uniformity of Cell Shape' real
@attribute 'Marginal Adhesion' real
@attribute 'Single Epithelial Cell Size' real
@attribute 'Bare Nuclei' real
@attribute 'Bland Chromatin' real
@attribute 'Normal Nucleoli' real
@attribute 'Mitoses' real
@attribute 'class' { 2, 4}
@data
1000025,5,1,1,1,2,1,3,1,1,2
1002945,5,4,4,5,7,10,3,2,1,2
1015425,3,1,1,1,2,2,3,1,1,2
1016277,6,8,8,1,3,4,3,7,1,2
```

Con el dataset modificado, lo abrimos con Weka y, lo primero que debemos de hacer es borrar el atributo '*Sample code number'*, ya que no nos es necesario.

Empezaremos con regresión logística. Para ello, elegimos el clasificador */Weka/classifiers/functions/logistic*. Se elige un numero de iteraciones -1 para que, así, haga las máximas posibles.

```
Correctly Classified Instances         677               96.8526 %
Incorrectly Classified Instances        22               3.1474 %
Kappa statistic                          0.9303
Mean absolute error                      0.0446
Root mean squared error                  0.1523
Relative absolute error                  9.8788 %
Root relative squared error             32.0527 %
Total Number of Instances              699

=== Detailed Accuracy By Class ===

                 TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                   0.976     0.046     0.976     0.976     0.976      0.996     2
                   0.954     0.024     0.954     0.954     0.954      0.996     4
Weighted Avg.      0.969     0.038     0.969     0.969     0.969      0.996

=== Confusion Matrix ===

   a   b   <-- classified as
 447  11 |   a = 2
  11 230 |   b = 4
```

Como vemos, solo clasifica bien el **96.8526%** del dataset. Lo que significa que nuestro algoritmo clasifica mejor, ya que clasifica el 100%.

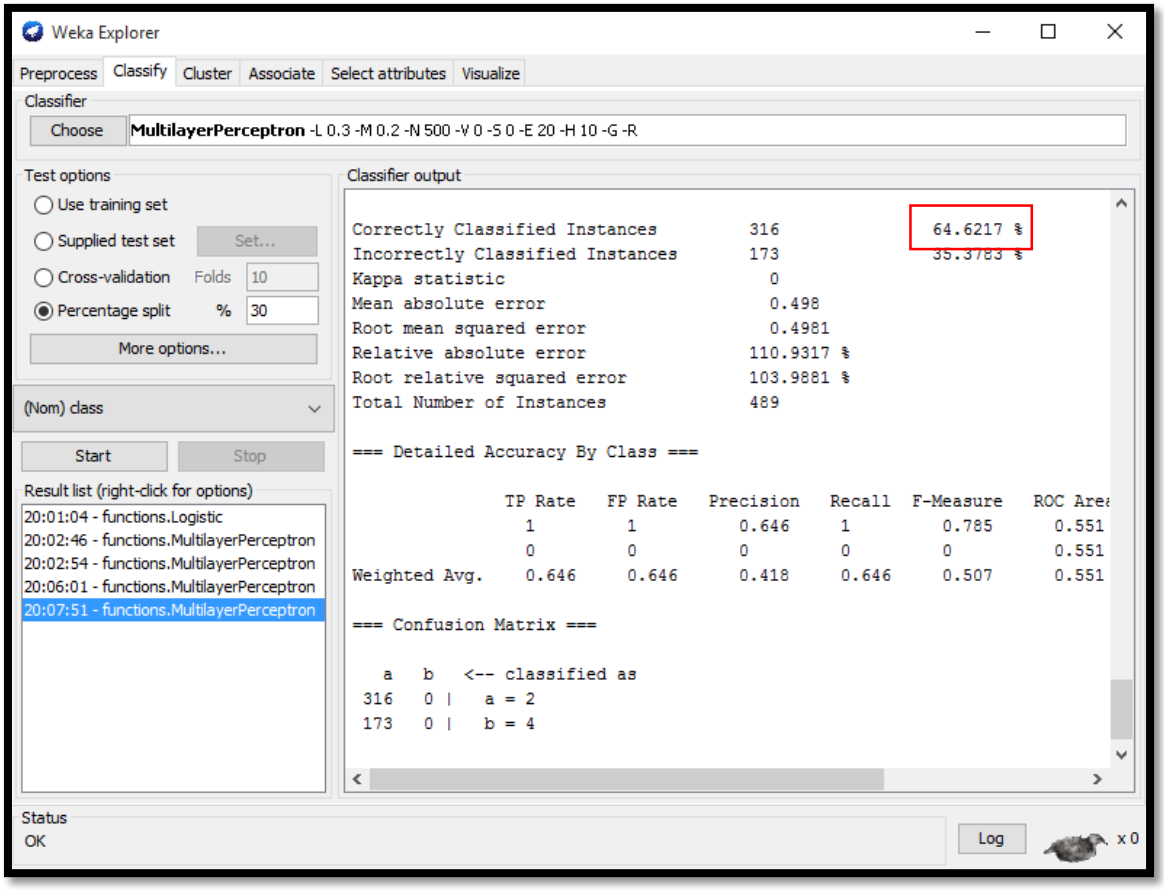
Ahora pasaremos a las **redes neuronales**. Aquí pondremos especial atención, ya que como recordaremos, se nos dio bastante mal, debido a las limitaciones que mostraba nuestro dataset y que octave no se ponía de nuestra parte con la ejecución.

Elegiremos el clasificador de preceptron, que es el que usa Weka para las redes neuronales.

```
Correctly Classified Instances         691               98.8555 %
Incorrectly Classified Instances         8                1.1445 %
Kappa statistic                          0.9748
Mean absolute error                      0.0206
Root mean squared error                  0.1032
Relative absolute error                  4.5487 %
Root relative squared error             21.7039 %
Total Number of Instances              699

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.985    0.004      0.998     0.985     0.991      0.995     2
                 0.996    0.015      0.972     0.996     0.984      0.995     4
Weighted Avg.    0.989    0.008      0.989     0.989     0.989      0.995

=== Confusion Matrix ===

   a   b   <-- classified as
 451   7 |   a = 2
   1 240 |   b = 4
```

Efectivamente, clasifica bien el **98%** del set de entrenamiento. Mi primer pensamiento fue que era normal, ya que lo lanzamos con 500 iteraciones, algo a lo que octave parecía ponerme problema y darme errores en los cálculos (Conclusiones de redes neuronales: Nanvalores.)

Seguí trasteando un poco con las redes neuronales. Consideré el usar un Split del 70%(100% - 30% como se muestra en la figura), tal y como usaba yo en mi algoritmo.



Solo clasificó bien el 64% de los ejemplos, un 10% menos de lo que clasificaba mi algoritmo. Eso quiere decir que hacer un Split en este dataset es más determinante de lo que creía.

Directamente fui a probar a hacer la red neuronal con todo el dataset, no solamente con el 70%. Efectivamente, cuando lo ejecuté con este cambio, clasificaba bien el 100% del dataset para cualquier valor de los que probé de lambda.

Conclusiones

Gracias a Weka pude resolver el problema que tenía de mal clasificación. Creo que es bastante beneficioso que, después de desarrollar un algoritmo, puedas comprobar su correcto funcionamiento con una herramienta en la que, con unas pocas configuraciones, tengas los resultados.

Referencias

ACCESO A SPECFUN: https://directory.fsf.org/wiki/Octave-specfun

ACCESO A LIBSVM: https://www.csie.ntu.edu.tw/~cjlin/libsvm/

ACCESO A DOC CON INSTRUCCIONES PARA SVMTOY: https://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0ahUKEwim6JSfsffKAhVJdCwKHefTD2MQFggzMAI&url=https%3A%2F%2Fwww.csie.ntu.edu.tw%2F~cjlin%2Flibsvm%2Fotherdocuments%2Fwong_svmtoy.doc&usg=AFQjCNGDQUdE0JnOL93hWjalOu-X8_JCRg

ACCESO A WEKA: http://www.cs.waikato.ac.nz/ml/weka/