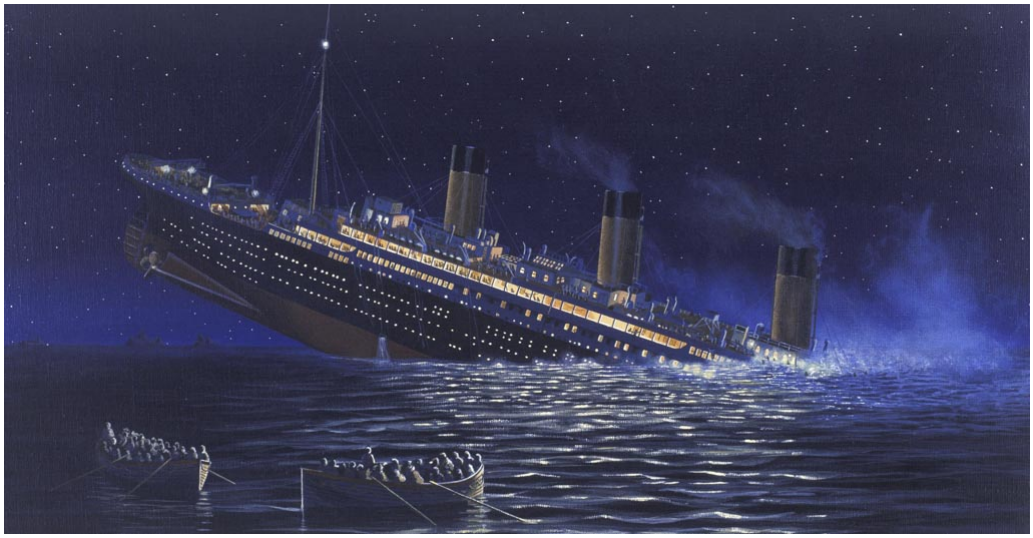
Universidad Complutense de Madrid

Facultad de Informática



Desarrollo y aplicación de algoritmos de aprendizaje automático para clasificar la supervivencia en el hundimiento del Titanic

Autores **David Ortiz Fernández**
Andrés Ortiz Loaiza

Aprendizaje Automático

Grado en Ingeniería de Computadores

Septiembre de 2018

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivaciones

El hundimiento del RMS Titanic es uno de los naufragios más famosos de la historia. El 15 de abril de 1912, durante su viaje inaugural desde Southampton a Nueva York, el Titanic se hundió después de colisionar con un *iceberg*, muriendo en esta tragedia un total de 1502 de 2224, entre pasajeros y tripulantes. Este accidente conmocionó a la comunidad internacional y condujo a un incremento en las regulaciones de seguridad para los buques transatlánticos.

Una de las razones por las que el naufragio provocó tal pérdida de vidas fue que no había suficientes botes salvavidas para los pasajeros y la tripulación. Aunque hubo algún elemento de suerte involucrado en sobrevivir al hundimiento, algunos grupos de personas tenían más probabilidades de sobrevivir que otros.

Hace unos años, la plataforma *Kaggle* lanzo un reto a la comunidad que consistía en aplicar diferentes algoritmos de aprendizaje automático y técnicas de minería de datos, para entrenar modelos que fueran capaces de predecir con el mayor porcentaje de aciertos posibles la supervivencia de los pasajeros del Titanic.

En este proyecto, se aplicarán y desarrollarán sobre un *dataset* disponible a través de la plataforma *Kaggle*, diferentes algoritmos de aprendizaje automático desarrollados en Octave, para predecir qué pasajeros sobrevivieron a la tragedia, enfocando para ello el problema como un problema de aprendizaje supervisado, y dentro de este ámbito como un problema de clasificación binaria.

1.2. Objetivos de este proyecto

Los objetivos específicos de este proyecto, serán los siguientes:

- Transformación y preparación de los datos obtenidos del repositorio disponible en la siguiente URL: https://www.kaggle.com/c/titanic/data.
- Desarrollo y aplicación de los diferentes algoritmos de aprendizaje supervisado estudiados durante el curso, enfocando el estudio como un problema de clasificación, utilizando para ello el entorno y lenguaje de programación GNU Octave.

- Estudio de los algoritmos desarrollados y aplicados.
- Toma de resultados de los diferentes algoritmos desarrollados.
- Análisis de los resultados obtenidos.
- Planteamiento de conclusiones finales.

1.3. Fases del trabajo

El proyecto se divide en las siguientes fases, con el fin de cumplir los objetivos arriba mencionados:

1. Se tratarán y preprocesarán los datos obtenidos en el repositorio mencionado, y se guardarán en el formato *csv*, compatible con Octave.
2. Se desarrollará un código en Octave de las diferentes funciones que nos permitan aplicar el algoritmo de regresión logística sobre los datos transformados.
3. Se desarrollará un código en Octave de las diferentes funciones que nos permitan aplicar el algoritmo de redes neuronales sobre los datos transformados.
4. Se desarrollará un código en Octave de las diferentes funciones que nos permitan aplicar el algoritmo SVM (*Support Vector Machines*) sobre los datos transformados.
5. Se realizará una toma de resultados aplicando los algoritmos desarrollados.
6. Análisis de los datos obtenidos con cada uno de los algoritmos aplicados.
7. Por último se expondrán las conclusiones.

Capítulo 2

Preprocesamiento de datos

En este capítulo se explican los datos iniciales que nos hemos encontrado en el repositorio de Kaggle. Y las técnicas utilizadas para transformarlo en los dataset que vamos a utilizar para nuestro proyecto

2.1. Datos originales

Tras descargar los archivos del repositorio mencionado y abrirlos, nos encontramos que no estaban preparados para ser utilizados directamente con Octave. Hay un total de 1309 ejemplos de entrenamiento divididos en dos archivos, unos destinados a entrenar y otros destinados a test. Decidimos juntar estos datos en un solo dataset, para nosotros hacer posteriormente las divisiones pertinentes. Los atributos originales que se recogían eran los siguientes:

- PassengerId: una manera de numerar nuestro dataset por filas.
- Survived: es el resultado real, es decir si vivieron o se ahogaron.
- Pclass: atributo que indica la clase en la que viajaba cada persona.
- Name: Nombre del pasajero
- Sex: hombre o mujer
- Age: edad del pasajero.
- SibSp: indica con valor numérico si tiene pareja a bordo.
- Parch: número de descendientes o padres a bordo.
- Ticket: número de ticket.
- Fare: tarifa que pagaron por el billete.
- Cabin: algunos pasajeros tenían una cabina asignada.
- Embarked: lugar en el que embarco, solo tiene tres valores posibles

2.2. Obtencion del primer dataset

Tras codificar un notebook en phyton hemos transformado los datos iniciales. Para ello eliminamos atributos que consideramos que no eran parte importante del dataset, el primero de ellos el ID, ya que era un atributo añadido por la propia plataforma. Después eliminamos el nombre, debido a su intrascendencia y el número de ticket, ya que en algunos casos no se registraba este valor. Lo siguiente fue dar un rango de valores a cada atributo no numérico.

Nos quedamos así con un total de 7 atributos numéricos por cada ejemplo de entrenamiento y la solución, en el mismo dataset. El nombre de este dataset es data.csv.

2.3. Obtención del segundo dataset

Para generar un nuevo atributo, decidimos aplicar técnicas de minería de datos, y conseguir el título asociado a cada persona, lo cual distinguiría su clase social, igualmente tras esto decidimos darle un rango de valores como en el dataset que obtuvimos en primera instancia. Por último se agrego este dato al anterior dataset y se le dio el nombre de TITANIC2.csv. Se realizo también el uso de las librerías de distintos algoritmos ya implementados, para hacer una estimación de los posibles resultados que podíamos esperar al desarrollar nosotros nuestros propios algoritmos. El código del notebook que genera ambos dataset se puede ver en las figuras que siguen.

```
# Andrés Ortiz
# David Ortiz.
import pandas as pd
import numpy as np
from sklearn import metrics
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
from sklearn import cross_validation
from sklearn.ensemble import RandomForestClassifier , AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
%matplotlib inline
```

```
titanic_train = pd.read_csv('train.csv') # For machine learning models.
titanic_test = pd.read_csv('test.csv') # To see how the model performs.
gender = pd.read_csv('gender_submission.csv')


#titanic_test
```

```
titanic_train
# Survival survival
# pclass ticket class -1st Upper - 2 Middle - 3 Lower
# sex Sex
# Age Age in years
# sibsp # of siblings - Sibling ( brother , sister , stepbrother , stepsister) - Spouse (wife,husband)
# parch # of parents - Parent ( mother ,father) - Child ( gaughter , son ) -
# ticket Ticket number
# fare Passenger fare
# cabin Cabin number
# embarked Port of Embarkation
```

FIGURA 2.1: Código del notebook.

```
titanic_train['Survived'].mean()
# The mean of survivors
```

```
0.3838383838383838
```

```
titanic_train.groupby('Pclass').mean()
#People who belong to an upper class survived more than others.
# People who belong to the lower class is youger than others.
# Then Pclass is an important feature to determinate the results.
```

	PassengerId	Survived	Age	SibSp	Parch	Fare
Pclass						
1	461.597222	0.629630	38.233441	0.416667	0.356481	84.154687
2	445.956522	0.472826	29.877630	0.402174	0.380435	20.662183
3	439.154786	0.242363	25.140620	0.615071	0.393075	13.675550

```
titanic_train.groupby('Sex').mean()
# In general females survived much more than males.
```

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
Sex							
female	431.028662	0.742038	2.159236	27.915709	0.694268	0.649682	44.479818
male	454.147314	0.188908	2.389948	30.726645	0.429809	0.235702	25.523893

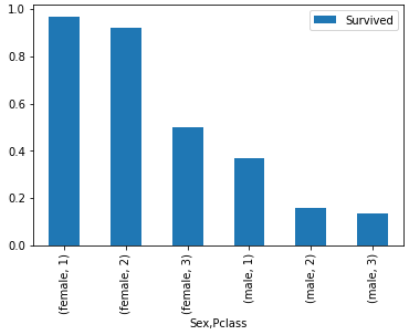
```
sex_pclass_groupby = titanic_train.groupby(['Sex','Pclass']).mean()
# If we group them by Sex and Pclass we can see than most females
# of upper and mid class survived.

del sex_pclass_groupby['PassengerId']
del sex_pclass_groupby['Age']
del sex_pclass_groupby['SibSp']
del sex_pclass_groupby['Parch']
del sex_pclass_groupby['Fare']

sex_pclass_groupby.plot.bar()

# Sex is a determinat feature.
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x23d393c66a0>
```




```
parch_group = titanic_train.groupby(['Parch','Pclass','Sex']).mean()
parch_group
```

FIGURA 2.2: Código del notebook.

```
titanic_train['Survived'].mean()
# The mean of survivors
```

```
0.3838383838383838
```

```
titanic_train.groupby('Pclass').mean()
#People who belong to an upper class survived more than others.
```

```
# Let's extract the title of the names to avoid the name colums from the analysis.
# We will use a regular expression to do it.
titanic_train['Title'] = titanic_train["Name"].str.extract(' ([A-Za-z]+)\.',expand=False)
titanic_test['Title'] = titanic_test["Name"].str.extract(' ([A-Za-z]+)\.',expand=False)

# Convert to rare type the strange values.
titanic_train['Title'] = titanic_train['Title'].replace(['Lady','Countess','Capt','Col',\
                                                         'Don','Dr','Major','Rev','Sir','Jonkheer','Dona'],'Rare')
titanic_test['Title'] = titanic_test['Title'].replace(['Lady','Countess','Capt','Col',\
                                                       'Don','Dr','Major','Rev','Sir','Jonkheer','Dona'],'Rare')

# Replace values that mean the same thing.
titanic_train['Title'] = titanic_train['Title'].replace('Mlle','Miss')
titanic_train['Title'] = titanic_train['Title'].replace('Ms','Miss')
titanic_train['Title'] = titanic_train['Title'].replace('Mme','Mrs')

titanic_test['Title'] = titanic_test['Title'].replace('Mlle','Miss')
titanic_test['Title'] = titanic_test['Title'].replace('Ms','Miss')
titanic_test['Title'] = titanic_test['Title'].replace('Mme','Mrs')




# Now we have to convert some string values to int values
encoding = LabelEncoder()
encoding2 = LabelEncoder()
encoding3 = LabelEncoder()
encoding4 = LabelEncoder()
encoding5 = LabelEncoder()
encoding6 = LabelEncoder()
```

```
encoding.fit(titanic_train['Sex'].values)
encoding2.fit(titanic_test['Sex'].values)
encoding3.fit(titanic_test['Embarked'].values)
Embarked2 = pd.factorize(titanic_train['Embarked'])
Cabin1 = pd.factorize(titanic_train['Cabin'])
Cabin2 = pd.factorize(titanic_test['Cabin'])
titanic_train['Embarked']=Embarked2[0]
titanic_train['Cabin']=Cabin1[0]
titanic_test['Cabin']=Cabin2[0]
encoding5.fit(titanic_train['Title'].values)
encoding6.fit(titanic_test['Title'].values)
```

```
LabelEncoder()
```

```
Sex = encoding.transform(titanic_train['Sex'].values)
Sex2 = encoding2.transform(titanic_test['Sex'].values)
Embarked = encoding3.transform(titanic_test['Embarked'].values)
Title1 = encoding5.transform(titanic_train['Title'].values)
Title2 = encoding6.transform(titanic_test['Title'].values)
```

```
titanic_train['Sex'] = Sex
titanic_test['Sex'] = Sex2
titanic_test['Embarked'] = Embarked
titanic_train['Title'] = Title1
titanic_test['Title'] = Title2
```

```

```

```
titanic_train = titanic_train.fillna(titanic_train.mean())
titanic_test = titanic_test.fillna(titanic_test.mean())

features = titanic_train[[ 'Survived','Title',"Sex","Pclass","Age","SibSp",'Fare','Parch','Embarked']]
X = features
Y = titanic_train['Survived']


#X_train,X_test,y_train,y_test=train_test_split(X,Y,test_size=0.4,random_state=42)

#X_test

X_test = titanic_test[['Title',"Sex","Pclass","Age",'SibSp','Fare','Parch','Embarked']]
```

FIGURA 2.3: Código del notebook.

```
# Let's extract the title of the names to avoid the name colums from the analysis.
# We will use a regular expression to do it.
titanic_train['Title'] = titanic_train["Name"].str.extract(' ([A-Za-z]+)\.',expand=False)
titanic_test['Title'] = titanic_test["Name"].str.extract(' ([A-Za-z]+)\.',expand=False)

# Convert to rare type the strange values.
titanic_train['Title'] = titanic_train['Title'].replace(['Lady','Countess','Capt','Col',\
                                                         'Don','Dr','Major','Rev','Sir','Jonkheer','Dona'],'Rare')
titanic_test['Title'] = titanic_test['Title'].replace(['Lady','Countess','Capt','Col',\
                                                       'Don','Dr','Major','Rev','Sir','Jonkheer','Dona'],'Rare')

# Replace values that mean the same thing.
titanic_train['Title'] = titanic_train['Title'].replace('Mlle','Miss')
titanic_train['Title'] = titanic_train['Title'].replace('Ms','Miss')
titanic_train['Title'] = titanic_train['Title'].replace('Mme','Mrs')

titanic_test['Title'] = titanic_test['Title'].replace('Mlle','Miss')
titanic_test['Title'] = titanic_test['Title'].replace('Ms','Miss')
titanic_test['Title'] = titanic_test['Title'].replace('Mme','Mrs')




# Now we have to convert some string values to int values
encoding = LabelEncoder()
encoding2 = LabelEncoder()
encoding3 = LabelEncoder()
encoding4 = LabelEncoder()
encoding5 = LabelEncoder()
encoding6 = LabelEncoder()
```

```
encoding.fit(titanic_train['Sex'].values)
encoding2.fit(titanic_test['Sex'].values)
encoding3.fit(titanic_test['Embarked'].values)
Embarked2 = pd.factorize(titanic_train['Embarked'])
Cabin1 = pd.factorize(titanic_train['Cabin'])
Cabin2 = pd.factorize(titanic_test['Cabin'])
titanic_train['Embarked']=Embarked2[0]
titanic_train['Cabin']=Cabin1[0]
titanic_test['Cabin']=Cabin2[0]
encoding5.fit(titanic_train['Title'].values)
encoding6.fit(titanic_test['Title'].values)
```

```
LabelEncoder()
```

```
Sex = encoding.transform(titanic_train['Sex'].values)
Sex2 = encoding2.transform(titanic_test['Sex'].values)
Embarked = encoding3.transform(titanic_test['Embarked'].values)
Title1 = encoding5.transform(titanic_train['Title'].values)
Title2 = encoding6.transform(titanic_test['Title'].values)
```

```
titanic_train['Sex'] = Sex
titanic_test['Sex'] = Sex2
titanic_test['Embarked'] = Embarked
titanic_train['Title'] = Title1
titanic_test['Title'] = Title2
```

```
titanic_train = titanic_train.fillna(titanic_train.mean())
titanic_test = titanic_test.fillna(titanic_test.mean())
```

```
features = titanic_train[[ 'Survived','Title',"Sex","Pclass","Age","SibSp",'Fare','Parch','Embarked']]
X = features
Y = titanic_train['Survived']


#X_train,X_test,y_train,y_test=train_test_split(X,Y,test_size=0.4,random_state=42)

#X_test

X_test = titanic_test[['Title',"Sex","Pclass","Age",'SibSp','Fare','Parch','Embarked']]

X_test['Survived']=gender['Survived']

gender_test = X_test[['Survived','Title',"Sex","Pclass","Age","SibSp",'Fare','Parch','Embarked']]
#titanic_test.describe()
titanic_train.describe()

features
```

```
gender_test.to_csv("TITANIC1.csv", sep=',', encoding='utf-8',index=False)
features.to_csv("TITANIC2.csv", sep=',', encoding='utf-8',index=False)
```

FIGURA 2.4: Código del notebook.

Capítulo 3

Regresión logística regularizada

En este capítulo se expondrá la metodología, código y los resultados asociados al primer algoritmo de aprendizaje automático que hemos implementado. Debido a que hemos enfocado nuestro problema como un problema de aprendizaje supervisado, y dentro de este tipo de problemas como uno de clasificación binaria, será la regresión logística regularizada nuestro primer algoritmo.

Dado que el objetivo de este proyecto es predecir la supervivencia en el Titanic, no se trata de un problema de clasificación multivariable, sino de uno de clasificación binaria.

3.1. Normalización de los atributos

Antes de proceder con el algoritmo, hemos separado el *dataset* en atributos y resultado, y se han creado con esto las matrices correspondientes.

A la hora de plantearnos normalizar los atributos, decidimos no hacerlo puesto que los valores de estos se encuentran en rangos próximos y normalizar podría conducir a errores, igualmente no se ha visto ningún orden al inspeccionar el datatest original, por lo que no ha parecido innecesario reordenarlos aleatoriamente.

3.2. Entrenamiento y predicción

A continuación, se describirán las diferentes fases en las que hemos dividido el proceso de entrenamiento en el script principal, en el que hemos implementado nuestro algoritmo de regresión logística regularizada:

- Primera fase: primero de todo hemos mostrado el coste inicial, eligiendo un valor para lambda igual a 0, de esta manera no se incluye la regularización, y unos valores iniciales de Θ también igual a 0. Para ello se ha desarrollado una función que calcula el coste y el gradiente para unos valores de Θ dados. El código de esta función aparece en la figura 3.1, y el código de la función sigmoide utilizada por esta se adjunta en la figura 3.2.

  Los valores obtenidos para el coste inicial con los valores mencionados ha sido: 0.69, y para obtenerlo se ha utilizado el total del dataset.

```
function [J, grad] = costeReg(theta, X, y, lambda)
m = length(y);
J = 0;
grad = zeros(size(theta));

h = fsigmoide(X*theta);

stheta = theta(2:size(theta));
thetaReg = [0;stheta];

% J = (1/m)*sum(-y .* log(h) - (1 - y) .* log(1-h));
J = (1/m)*(-y'* log(h) - (1 - y)'*log(1-h))+(lambda/(2*m))*thetaReg'*thetaReg;

grad = (1/m)*(X'*(h-y)+lambda*thetaReg);

endfunction
```

FIGURA 3.1: Código de la función de coste y gradiente.

```
function g = fsigmoide(z)
  %la función se puede aplicar a une escalar, matriz o vector
  g = 1./ (1 + e.^-z);
endfunction
```

FIGURA 3.2: Código de la función sigmoide.

Tras cambiar el valor del parámetro de regularización lambda a 2, y de esta manera incluir la regularización, hemos aplicado descenso de gradiente gracias a la función fminunc.m con un total de 400 iteraciones. De esta manera se han conseguido los valores óptimos de Θ para el coste mínimo. Los valores obtenidos así para el coste y tras aplicar descenso de gradiente ha sido de 0.38.

También se ha decidido implementar y utilizar en esta fase una función que calcule el porcentaje de aciertos, la cual se puede ver en la figura 3.3. Tras aplicarla sobre nuestro modelo con el total del dataset, se ha conseguido un acierto del 85.48 %, este porcentaje es sobre los mismos datos con los que hemos entrenado el modelo, por lo cual carecen de validez. Además, se ha calculado el tiempo en entrenar el modelo y alcanzar convergencia, siendo este de 0.08 segundos.

```
function p = porcentaje(theta, X, y)
  m = size(X, 1); % numero de ejemplos de entrenamiento
  p = zeros(m, 1);

  p = (fsigmoide(X*theta) >= 0.5);

  printf('Porcentaje de aciertos (accuracy): %f\n', mean(double(p == y)) * 100);
  p = mean(double(p == y)) * 100;
endfunction
```

FIGURA 3.3: Código de la función que calcula el porcentaje de aciertos.

El código del script principal asociado a la primera fase aparece en la figura 3.9.

- Segunda fase: llegados a este punto dividimos el dataset en dos grupos, uno para entrenar del 70 %, y otro para test, del 30 %. Nuevamente aplicamos descenso de gradiente y entrenamos nuestro modelo con los datos de entrenamiento esta vez. Al calcular el porcentaje de aciertos sobre el 30 % que suponían los datos de test, obtuvimos un porcentaje de aciertos de 95.93 %. El código del script principal asociado a esta fase se muestra en la figura 3.10.

- Tercera fase: en esta fase hemos aplicado cross-validation a la regresión logística, para ello hemos dividido el dataset en tres partes, una contiene los datos de entrenamiento (60 %), otra los de validación cruzada (20 %), y otra los datos para test(20 %). Para aplicar cross-validation, hemos desarrollado la función validacion.m, cuyo código aparece en la figura 3.8, la cual aplica el descenso de gradiente variando el valor de lambda, y calcula los valores Θ óptimos, con estos valores calculamos el valor del error de validación y el error de entrenamiento para cada lambda, recordemos que para ello debemos aplicar el cálculo del coste sin el parámetro de regularización, es decir con lambda igual a cero.

  En la tabla 3.1 se muestran los valores de error de entrenamiento y validación para cada lambda.

Lambda	Error entrenamiento	Error validación
0.01	0.4529	0.3400
0.03	0.4529	0.3401
0.1	0.4529	0.3405
0.3	0.4529	0.3415
1	0.4531	0.3452
3	0.4545	0.3549
10	0.4634	0.3839
30	0.4907	0.4394

TABLA 3.1: Resultados de error de entrenamiento y validación para cada valor de lambda en regresión logística.

  Con el valor de lambda seleccionado, que es el que proporciona menor error de validación, calculamos los aciertos sobre los datos de test. Una correcta regularización nos ayudará a evitar el problema del overfitting. Además, esta función muestra y crea las curvas de aprendizaje y la variación del error en función del valor de lambda. Un vez aplicada la función, obtuvimos un valor de lambda óptimo de 0.01, y un porcentaje de aciertos sobre los datos de test de 96.56 %, el cual consideramos que tiene mayor validez que los anteriores.

  En cuanto a las curvas de aprendizaje obtenidas se muestran en la figura3.4

  También hemos mostrado la gráfica de la evolución de los errores de entrenamiento y validación en función del parámetro de regularización, la cual se muestra en la figura 3.5.

- Cuarta fase: hemos desarrollado funciones para el cálculo del recall y precisión, como una alternativa al porcentaje de aciertos (accuracy) para evaluar nuestro clasificador, así como

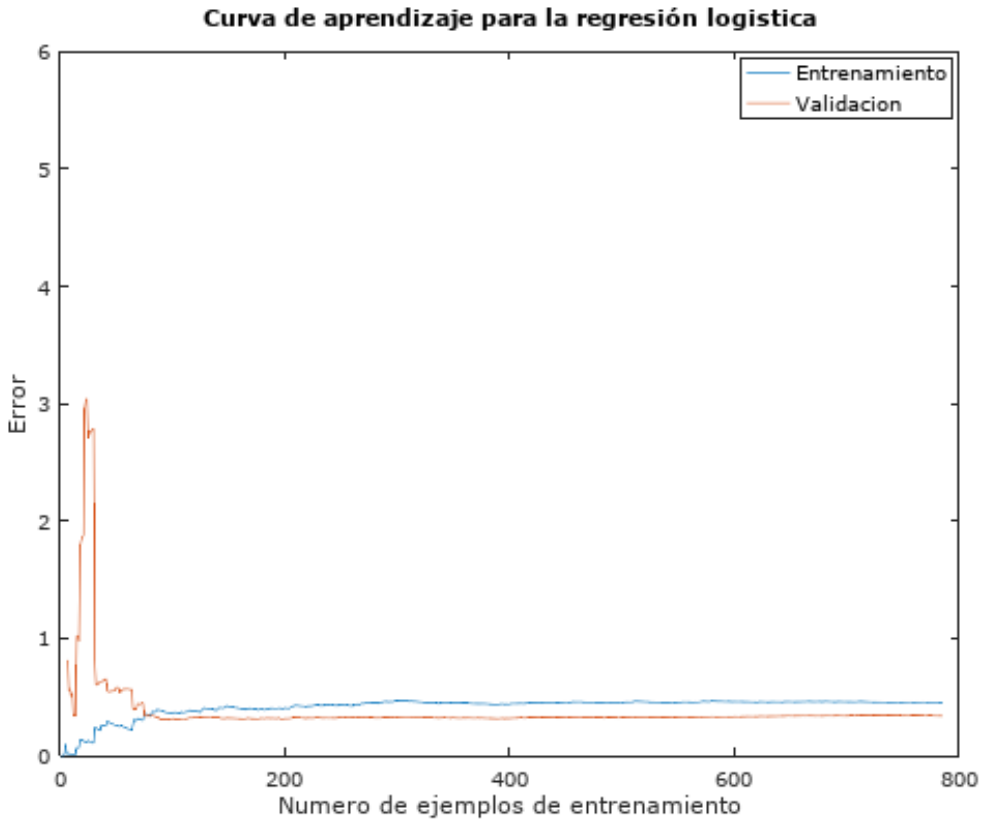


FIGURA 3.4: Curvas de aprendizaje de nuestro modelo.

el computo del threshold. Para ello se han implementado la función precisionrecall.m cuyo código aparece en la figura 3.6 y la función threshold.m cuyo código aparece en la figura. 3.7. Con el valor de precision estamos obteniendo de todos los pasajeros que se les ha clasificado como ahogados, cuantos se ahogaron realmente, y con el de recall se obtienen de todos los pasajeros que se ahogaron, a cuantos se les ha clasificado correctamente. Los valores de precisión y recall han sido de 97.80 % y 93.68 % respectivamente. Para el calculo de estos valores ha sido necesario calcular el threshold optimo previamente.

- Quinta fase: como adelantamos en el capítulo 2, añadimos un parámetro adicional, el cual no aparecía originalmente en el dataset, nuestro objetivo era ver como influenciaba este parámetro en el porcentaje de predicción alcanzado con cross-validation, así como se verá en el capítulo de conclusiones intentar paliar un posible problema de alto sesgo. Los resultados al repetir la tercera fase con este nuevo parámetro que indica el título de la persona, como ya hemos mencionado en el capítulo anterior, y tras haber entrenado nuestro modelo con cross-validation ha sido de 92.74 %, siendo nuevamente el valor óptimo de lambda de 0.01. Nuestro clasificador tiene una precision de 89.79 % y un recall de 92.63 % en este caso.

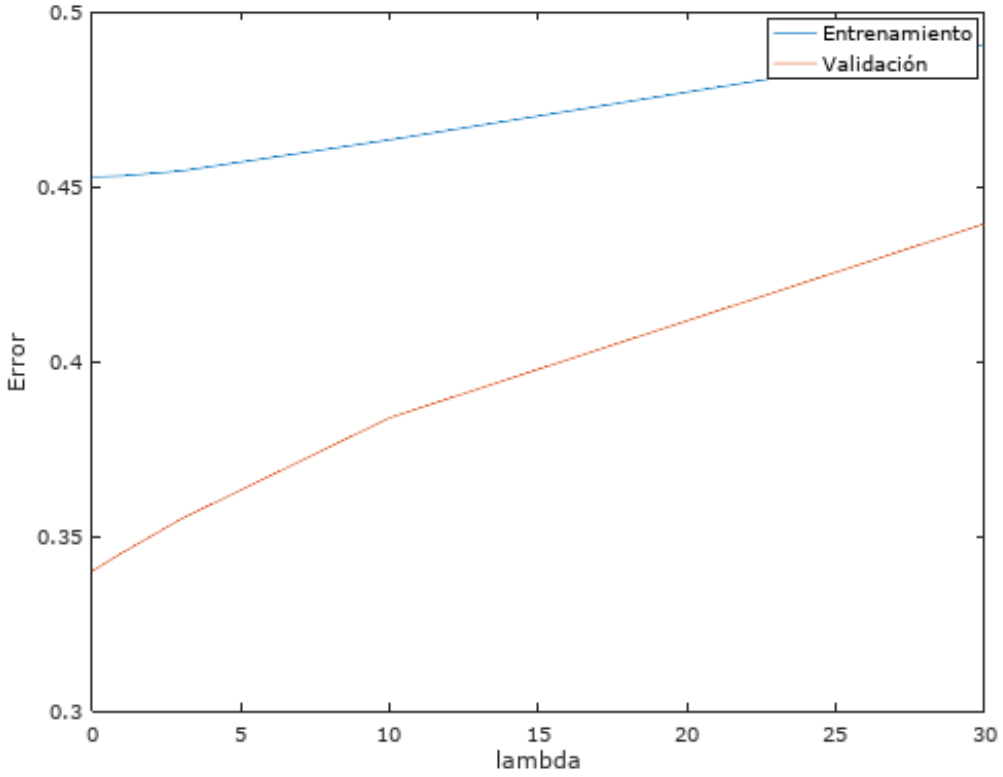


FIGURA 3.5: Curvas de evolución de los costes de entrenamiento y validación.

```
function [precision, recall] = precisionrecall(theta, Xtest, ytest, threshold)

    resultados = fsigmoide(Xtest*theta);
    %calculamos positivos reales en el test
    positivosReales = ytest == 1;

    positivosPred = resultados >= threshold;
    %alamacenamos un 0 o un 1 dependiendo de si suman dos o no
    truepos = (positivosReales + positivosPred == 2);

    if (positivosPred == 0)
        precision = 1;
    else
        %calculo de la precision
        precision = sum(truepos) / sum(positivosPred);
    endif
    %calculo del recall
    recall = sum(truepos) / sum(positivosReales);
endfunction
```

FIGURA 3.6: Código de la función que calcula la precisión y el recall para la regresión logística.

```
function th = threshold(theta, X, y)
  candidato = [0 : 0.01 : 1];
  maxf1 = 0;
  %para candidato de ser el threshold optimo
    for i = 1:columns(candidato)
    %calculamos para cada candidato a ser el threshold su precision y su recall
       [precision, recall] = precisionrecall(theta, X, y, candidato(:,i));
       f1score = 2 * ((precision * recall) / (precision + recall));
    %elegimos el de mayor score
       if(maxf1 < f1score)
          maxf1 = f1score;
          th = candidato(:,i);
        endif
    endfor
endfunction
```

FIGURA 3.7: Código de la función que calcula threshold óptimo.

```
function[lambda_vec,error_train,error_val]=validacion(X,y,Xval,yval,Xtest,ytest)
  %variables utiles
  lambda_vec = [0 0.01 0.03 0.1 0.3 1 3 10 30]';
  maxpercentage = 0;
  minerrorval = realmax;
  bestlambda = 0;

  error_train = zeros(length(lambda_vec), 1);
  error_val = zeros(length(lambda_vec), 1);
  options = optimset('GradObj', 'on', 'MaxIter', 400);
  [m, n] = size(X);
  initial_theta = zeros(n , 1);

  for i = 1:length(lambda_vec)
      printf('Entrenando con lambda_vec %d/%d \n', i, length(lambda_vec));
      lambda = lambda_vec(i);
      %Para cada lambda entrenamos y calculamos los theta y el coste
      [theta,cost]=fminunc(@(t)(costeReg(t, X,y,lambda)),initial_theta,options);

%Calculamos el error con lambda 0, tiene que calcularse sin el termino
%regularizado
    error_train(i) = costeReg(theta,X,y, 0);
    error_val(i) = costeReg(theta,Xval,yval,0);

      if (error_val(i) < minerrorval)
        minerrorval = error_val(i);
        bestlambda = lambda;
        besttheta = theta;
         bestcost = cost;
       endif
    endfor
  %tras cros validation calculamos los aciertos para el test
    p = porcentaje(besttheta, Xtest, ytest);
    printf('El lambda optimo encontrado es %f que ha clasificado correctamente
                                  el %f de los datos de test. \n', bestlambda, p);
 %pintado de seleccion de lambda
    figure(1);
    plot(lambda_vec, error_train, lambda_vec, error_val);
    legend('Entrenamiento', 'Validación');
    xlabel('lambda');
    ylabel('Error');
    printf('\n Presione Entrer para continuar.\n');
    pause;
 %calculo de curvas de aprendizaje
    for i = 1:m
        [theta, cost] = fminunc(@(t) (costeReg(t,X(1:i,:),y(1:i,:),bestlambda)),
                                                    initial_theta, options);
        [jtrain(i), grad] = costeReg(theta, X, y, 0);
        [jval(i), grad] = costeReg(theta, Xval, yval, 0);
    endfor
 %pintado de curva de aprendizaje
    figure(2);
    plot(1:m, jtrain, 1:m, jval);
    title('Curva de aprendizaje para la regresión logistica')
    legend('Entrenamiento', 'Validacion')
    xlabel('Numero de ejemplos de entrenamiento')
    ylabel('Error')
endfunction
```

FIGURA 3.8: Código de la función donde se implementa cross validation.

```
clear ;
close all ;
addpath("regresion");
warning('off','all');
%cargamos los datos del csv
data = csvread('data.csv');

% sacamos la feature ID de los datos y separamos las features de los label
%LEGENDA [Survived, PassengerId, Pclass, Sex(female=1), Age, SibSp, Parch,
%                Fare, Cabin, Embarked (S=1, C=2 , Q=3]
y = data(:,1);
X = data(:,3:end);
[m , n]=size(X);
%% ============ REGRESION LOGISTICA SOBRE EL 100% DEL DATASET==================
%añadimos columna de 1´s a las matrices
X = [ones(m, 1) X];

%A continuación, hemos aplicado regresión logística  sobre el 100% de los datos
printf('FASE 1: regresión logistica regularizada sobre el 100 del dataset. \n');
printf('Pulsa Enter para continuar...\n');
pause;
%inicializamos los valore theta a 0
thetainicial = zeros(n + 1, 1);
lambda = 0;
opciones = optimset('GradObj', 'on', 'MaxIter', 400);
% Coste y gradiente con theta iniciales
[cost, grad] = costeReg(thetainicial, X, y, lambda);
printf('->Función J de Coste para valores theta iniciales: %f\n', cost);
printf('Presione Enter para continuar.\n');
pause;

%Aplicacion del descenso de gradiente con fminunc
lambda = 2;
tic;
% Obtención de los valores óptimos de theta
[theta, cost] = fminunc(@(t)(costeReg(t, X, y, lambda)),thetainicial, opciones);
time = toc;

printf('->Coste minimo con valores lambda de 2 y MAXITER de 400 : %f\n', cost);
printf('->El tiempo para alcanzar convergencia a sido: %.2f segundos\n',time);

printf('->Porcentaje de aciertos sobre el total del dataset, tras entrenar con
                                                                    todo.');
p = porcentaje(theta, X, y);
printf('Presione Enter para continuar.\n');
pause;
printf('================================================================\n');
```

FIGURA 3.9: Código del flujo principal de la primera fase.

```
printf('FASE 2: regresión logistica regularizada sobre el 70 del dataset. \n');
printf('Pulsa Enter para continuar...\n');
dataTr = data(1:916,:);
dataTest = data(917:end,:);
y_tr = dataTr(:,1);
X_tr = dataTr(:,3:end);
y_test = dataTest(:,1);
X_test = dataTest(:,3:end);
%valores utiles
[m_tr , n_tr]=size(X_tr);
[m_test , n_test]=size(X_test);

%añadimos columna de 1´s a las matrices para theta 0
X_tr = [ones(m_tr, 1) X_tr];
X_test = [ones(m_test, 1) X_test];

%inicializamos los valore theta 0 y lambda a 2
thetainicial = zeros(n_tr + 1, 1);
lambda = 3;

%Cálculo del valor óptimo de los parámetros con fminunc y theta optimas
opciones = optimset('GradObj', 'on', 'MaxIter', 400);
tic;
[theta, cost]=fminunc(@(t)(costeReg(t,X_tr,y_tr,lambda)),thetainicial,opciones);
tiemporeg = toc;
printf('Precisión calculada tras entrenar el modelo con el 70 por ciento, y
                                                    tetear con el 30 por ciento');
p = porcentaje(theta, X_test, y_test);
```

FIGURA 3.10: Código del flujo principal de la segunda fase.

```
printf('FASE 3: entrenamiento con el 60 aplicando cross- validation. \n');
%cargamos los datos
data = csvread('data.csv');
%dividimos las features y los label y extraemos la feature ID
dataTr = data(1:785,:);
dataCV = data(786:1047,:);
dataTest = data(1048:end,:);
% sacamos el ID de los datos y separamos las features de los label (column 2)
y_cv = dataCV(:,1);
X_cv = dataCV(:,3:end);
y_tr = dataTr(:,1);
X_tr = dataTr(:,3:end);
y_test = dataTest(:,1);
X_test = dataTest(:,3:end);
%valores utiles
[m_tr , n_tr]=size(X_tr);
[m_cv , n_cv] = size(X_cv);
[m_test , n_test]=size(X_test);

%añadimos columna de 1´s a las matrices para theta 0
X_tr = [ones(m_tr, 1) X_tr];
X_cv = [ones(m_cv, 1) X_cv];
X_test = [ones(m_test, 1) X_test];

%inicializamos los valore theta 0 y lambda a 3
thetainicial = zeros(n_tr + 1, 1);
lambda = 3;
% Coste inicial y gradiente sobre el 60% de los datos
[cost, grad] = costeReg(thetainicial, X_tr, y_tr, lambda);
printf('->Coste para valores theta iniciales: %f\n', cost);
printf('Presione Enter para continuar.\n');
pause;

%Cálculo del valor óptimo de los parámetros con fminunc y theta optimas
opciones = optimset('GradObj', 'on', 'MaxIter', 400);
tic;
[theta, cost]=fminunc(@(t)(costeReg(t,X_tr,y_tr,lambda)),thetainicial,opciones);
tiemporeg = toc;

printf('->Coste minimo con valores lambda de 2 y MAXITER de 400 : %f\n', cost);
printf('Presione Enter para continuar.\n');
pause;

%Curvas de validacion y aplicacion de cross validation
[lambda_vec,error_train,error_val]=validacion(X_tr,y_tr,X_cv,y_cv,X_test,y_test)
printf('Presione Enter para finalizar.\n');
pause;
```

FIGURA 3.11: Código del flujo principal de la tercera fase.

Capítulo 4

Redes neuronales

En este capítulo se explicará la metodología que se ha seguido para implementar y aplicar el algoritmo de redes neuronales que hemos desarrollado. Para ello a lo largo del capítulo se irán explicando las diferentes funciones utilizadas y desarrolladas, así como el modo de proceder aplicando las diferentes técnicas para una estimación de la predicción más fiable vistas durante el curso.

Frente a la regresión logística, las redes neuronales ofrecen la flexibilidad de poder aprender todos los atributos simultáneamente.

4.1. Red neuronal de una capa oculta.

Como ocurrió con la regresión logística, antes de proceder con el algoritmo de redes neuronales hemos separado el *dataset* en atributos y resultado, y se han creado con esto las matrices correspondientes. Igualmente no se han normalizado los atributos por las razones que se dieron en el capítulo anterior.

4.1.1. Arquitectura.

Para comenzar el desarrollo y aplicación de este algoritmo de clasificación de aprendizaje automático, nos plateamos utilizar una red neuronal de una única capa oculta, como las utilizadas para el desarrollo de las practicas del curso. Esta red neuronal consta de las siguientes capas:

- Una capa de entrada con 7 neuronas, una para cada atributo del conjunto de datos.
- Una única capa oculta con 9 neuronas, estando así en el rango de las neuronas de la capa de entrada.
- Una capa de salida con una única neurona de salida para clasificar a los individuos como supervivientes o fallecidos.

El problema de las redes neuronales, con una única capa oculta y pocas neuronas, es que son susceptibles del problema de *underfitting*, en contraposición de su alta eficiencia de computo. Intentando buscar eficiencia en el calculo y debido también a que es la que se ha usado durante el curso, decidimos elegir finalmente una red neuronal de una única capa oculta.

4.1.2. Inicialización de la red neuronal.

Para inicializar la red neuronal correctamente, desarrollamos una función que generase los pesos iniciales aleatorios, representados por dos vectores de pesos Θ con un valor pequeño cercano a cero, en el rango $[-epsilon\_ini, epsilon\_ini]$, siendo $epsilon = 0{,}12$, de esta manera aseguramos un correcto funcionamiento, al no ser 0, sino valores cercanos a este. El código de esta función aparece en la figura 4.1.

```
%función que inicialice una matriz de pesos T(l) con valores aleatorios en
%el rango [-epsilon_ini, epsilon_ini].
function W = pesosAleatorios(L_in, L_out)

W = zeros(L_out, 1 + L_in);
epsilon_ini = 0.12;

W = rand(L_out, 1 + L_in) * 2 * epsilon_ini - epsilon_ini;

endfunction
```

FIGURA 4.1: Código de la función presosAleatoriosNN

Nuestra red neuronal tendrá una función de activación sigmoidea por lo cual también se ha desarrollado el código de esa función, el cual aparece en la figura 4.2.

```
%función sigmoide
function g = fsigmoide(z)

g = 1.0 ./ (1.0 + exp(-z));

endfunction
```

FIGURA 4.2: Código de la función fsigmoide

4.1.3. Entrenamiento y precisión de la red neuronal.

Hemos dividido esta sección en varias fases, en función de las modificaciones que hemos ido realizando sobre los datos con los que hemos realizado el entrenamiento:

- Primera fase: Inicialmente hemos decidido entrenar nuestra red neuronal con el total de los datos del dataset, y así posteriormente comparar los resultados con los obtenidos utilizando cross-validation. Para entrenar nuestra red neuronal, hemos enrollado los parámetros Θ iniciales, los cuales se pasan como argumento a una función que implementa la propagación hacia delante y la retropropagación, esta función es costeRN, cuyo código aparece en la figura 4.11. El descenso de gradiente se llevará a cabo gracias a la función fmincg, proporcionada durante el curso.

  Para comprobar que la retropropagación está funcionando correctamente, hemos utilizado la función checkNNGradients, proporcionada en las practicas desarrolladas durante el curso, y

que compara los resultados obtenidos mediante el método de aproximación de la pendiente con el del obtenido en el cálculo de las derivadas parciales en la retropropagación, cuyo código aparece en la figura 4.3. Tras chequear si la retropropagación está actuando correctamente se puedo observar que así era, siendo la diferencia relativa de $1{,}92102e^{-011}$, la cual es lo suficientemente pequeña (menor que $1e^{-9}$).

```
function F = fsigmoideGradiente(z)

  F = zeros(size(z));

  F  = fsigmoide(z).*(1-fsigmoide(z));

endfunction
```

FIGURA 4.3: Código de la función fsigmoideGradiente.

Después de realizar el entrenamiento con el total del dataset, el cual duro 2.78 segundos, se obtuvieron los pesos óptimos que minimizaban el coste, siendo este coste mínimo de 0.49, tras un total de 100 iteraciones.

Tras el entrenamiento, se decidió chequear el porcentaje de aciertos de la red neuronal entrenada con el total de datos entrenado, sirviéndonos para ello de las funciones mean y double de Octave. De esta manera desarrollamos la función que aparece en la figura 4.4 El porcen-

```
function[acc]= accuracyNN(theta, X, y, num_entradas, num_ocultas, num_etiquetas)
    m=size(X,1);
    num_fallos = 0;
    p = zeros(size(X, 1), 1);

    h = forwardpropagation(X, theta, num_entradas, num_ocultas, num_etiquetas);
    [probs, p] = max(h,[],2);

    p = p .- 1;
    acc=(mean(double(p == y))) ;

endfunction
```

FIGURA 4.4: Código de la función accuracyNN

taje de aciertos obtenido fue de 66.08 %, lo cual nos sorprendió por su baja tasa de aciertos de clasificación correcta. Aun así, este porcentaje se ha obtenido al valorar los mismos datos usados para entrenar la red neuronal por lo que no debemos considerarlo un dato fiable de acierto.

El flujo principal hasta ahora aparece en la figura 4.12.

- Segunda fase: una vez entrenada la red con el total de datos y calculada su precisión con el total de estos, decidimos dividir los datos en dos grupos, uno para entrenar la red, que

suponen el 70 %, y un segundo grupo para calcular el índice de aciertos, que supone el 30 % del total de datos. Nuevamente replicamos todos los pasos dados hasta ahora, realizando los cambios pertinentes sobre el código manteniendo el valor del parámetro lambda en 3.

Los resultados del coste mínimo y de tiempo, nuevamente con 100 iteraciones, fueron 0.30 segundos y el coste mínimo ha sido: 0.465330 respectivamente. En cuanto a la precisión de la red sobre datos distintos a los destinados para el entrenamiento nos encontramos con un incremento en la predicción de la red, siendo esta de 75.57 %. Consideramos este porcentaje de acierto más valido que el conseguido al calcular la precisión sobre los datos con los que se ha entrenado, ya que estos no son los mismos. El código del flujo principal para esta parte aparece en la figura 4.13.

- Tercera fase: llegados a este punto separamos nuevamente los datos en tres grupos, para así aplicar la técnica de cross-validation. Los porcentajes elegidos para implementar esta técnica han sido de 60 % para los datos de entrenamiento, 20 % para los datos de validación y 20 % para los datos de test. Tras entrenar la red neuronal con los datos de entrenamiento, utilizamos los datos de cross-validation para seleccionar el parámetro lambda que proporcione un coste mínimo menor, tras aplicar la retropropagación. Así, se ha podido observar que el lambda óptimo ha sido 0.01. Tras cambiar el valor de lambda a este valor y volver a entrenar la red neuronal con los datos de entrenamiento, pero con este valor de lambda, calculamos nuevamente la precisión de nuestra red, como se ha visto, con el 20 % de los datos. El valor de precisión que se obtuvo de esta manera sobre los datos de test fue: 74.80 %. Tras este resultado consideramos este valor el más real en cuanto al porcentaje de acierto de nuestra red neuronal. En la figura 4.14 se puede observar el flujo de código principal asociado a esta parte.

  En esta fase hemos decidido mostrar la gráfica de la curva de aprendizaje, la cual se aprecia en la figura 4.15, la cual nos ayudara en las conclusiones finales. Recordemos que para una correcta creación de estas curvas, es necesario eliminar la regularización a la hora de calcular el coste. El código de esta parte del flujo principal aparece en la figura: 4.14.

  También hemos mostrado la gráfica de la evolución de los costes de entrenamiento y validación en función del parámetro de regularización, la cual se muestra en la figura 4.6 y cuyos valores se adjuntan en la tabla 4.1.

- Cuarta fase: Como siguiente paso, y como hicimos en la implementación de la regresión logística, calculamos el recall y la precisión de nuestra red neuronal una vez realizado el entrenamiento con cross-validation. Para ello se han desarrollado las funciones de precisionrecall.m con los cambios adecuados para redes neuronales y cuyo código se puede ver en la figura 4.7, lo mismo ha pasado con la función threshol.m, que aparece en la figura 4.8. La precision fue de 25.19 % y un recall de 100 %.

- Quinta fase: En este momento decidimos reducir la cantidad de atributos con los que entrenar a uno más, buscando un porcentaje de aciertos más elevado, al aplicar técnicas de

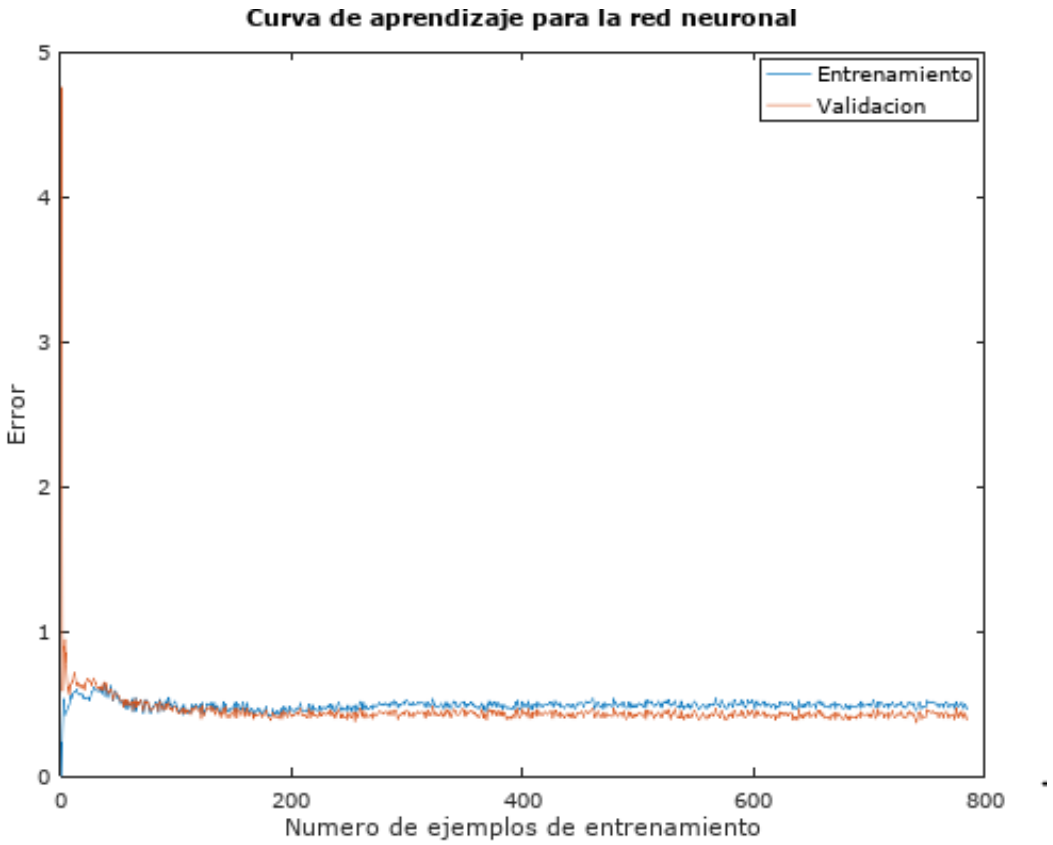


FIGURA 4.5: Curvas de aprendizaje de nuestra red neuronal

minería de datos sobre los atributos iniciales. De esta manera fue necesario cambiar la arquitectura de la red neuronal, puesto que las neuronas de la capa de entrada pasan a ser 8, al igual que el número de atributos. Igualmente aumentamos las neuronas de la capa oculta a 16. Reprodujimos los pasos seguidos en la tercera fase, intentando obtener en este caso un porcentaje de aciertos más elevado, pero los resultados fueron: un lambda óptimo de 1 y un porcentaje de aciertos de 63.74 % sobre los datos de test al aplicar cross-validation. El código desarrollado es exactamente igual al de la fase tres, salvo las excepciones mencionadas, por lo cual no se ha adjuntado en la memoria, aun así el script desarrollado y adjunto a esta memoria recibe el nombre de titanicNN2.m. Las gráficas de aprendizaje y lambda se muestran en la figura 4.9 y la figura 4.10 respectivamente. El valor de recall aumento a 36.25 % manteniendo la precisión al 100 % con este atributo extra.

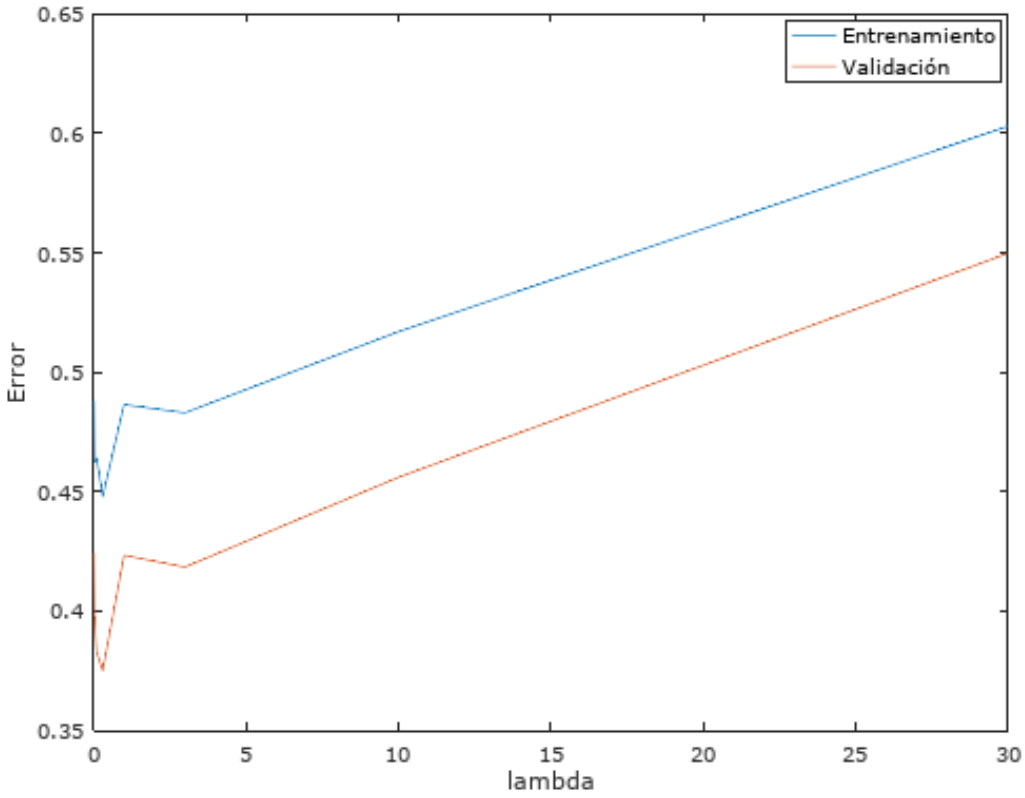


FIGURA 4.6: Curvas de evolución de los costes de entrenamiento y validación.

Lambda	Error entrenamiento	Error validación
0.01	0.4514	0.3750
0.03	0.5111	0.4474
0.1	0.4873	0.4202
0.3	0.4615	0.3808
1	0.5033	0.4413
3	0.4977	0.4319
10	0.5253	0.4583
30	0.6040	0.5493

TABLA 4.1: Resultados de error de entrenamiento y validación para cada valor de lambda en la red neuronal.

```
function [precision, recall] = precisionrecall(params_rn,X,y,threshold,
                                        num_entradas,num_ocultas,num_etiquetas)
   resultados = forwardpropagation(X, params_rn, num_entradas, num_ocultas,
                                                                num_etiquetas);
   %calculamos positivos reales en el test
    positivosReales = y == 1;
    positivosPred = resultados >= threshold;
        %almacenamos un 0 o un 1 dependiendo de si suman dos o no

    truepos = (positivosReales + positivosPred == 2);

  if (positivosPred == 0)
        precision = 1;
    else
        precision = sum(truepos) / sum(positivosPred);
    endif
    recall = sum(truepos) / sum(positivosReales);
endfunction
```

FIGURA 4.7: Código de la función que calcula la precisión y el recall para la red neuronal.

```
function th = threshold(theta, X, y, num_entradas, num_ocultas, num_etiquetas)
 %vector de candidatos
 candidato = [0 : 0.01 : 1]';
 maxf1 = 0;

for i = 1:rows(candidato)
  [precision, recall] = precisionrecall(theta, X, y, candidato(i),num_entradas,
                                                    num_ocultas, num_etiquetas);
   f1score = 2 * ((precision * recall) / (precision + recall));

        if(maxf1 < f1score)
            maxf1 = f1score;
            th = candidato(i);
        endif
    endfor
endfunction
```

FIGURA 4.8: Código de la función que calcula threshold óptimo para la red neuronal.

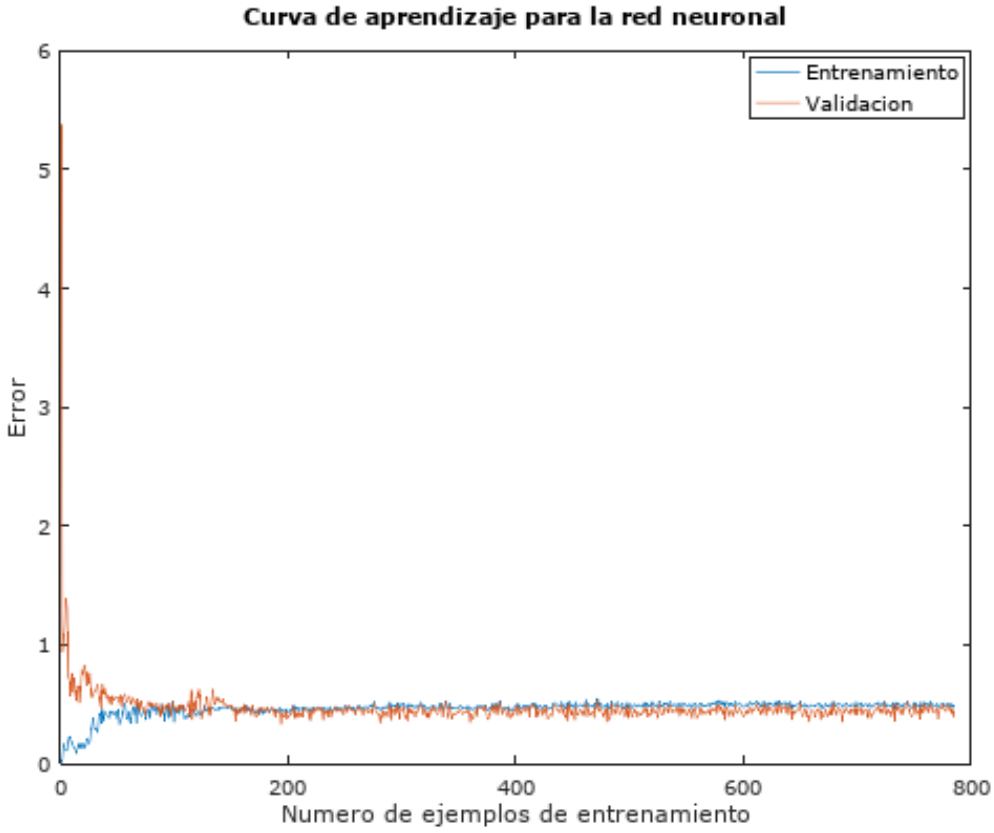


FIGURA 4.9: Curvas de aprendizaje de nuestra red neuronal con un atributo adicional.

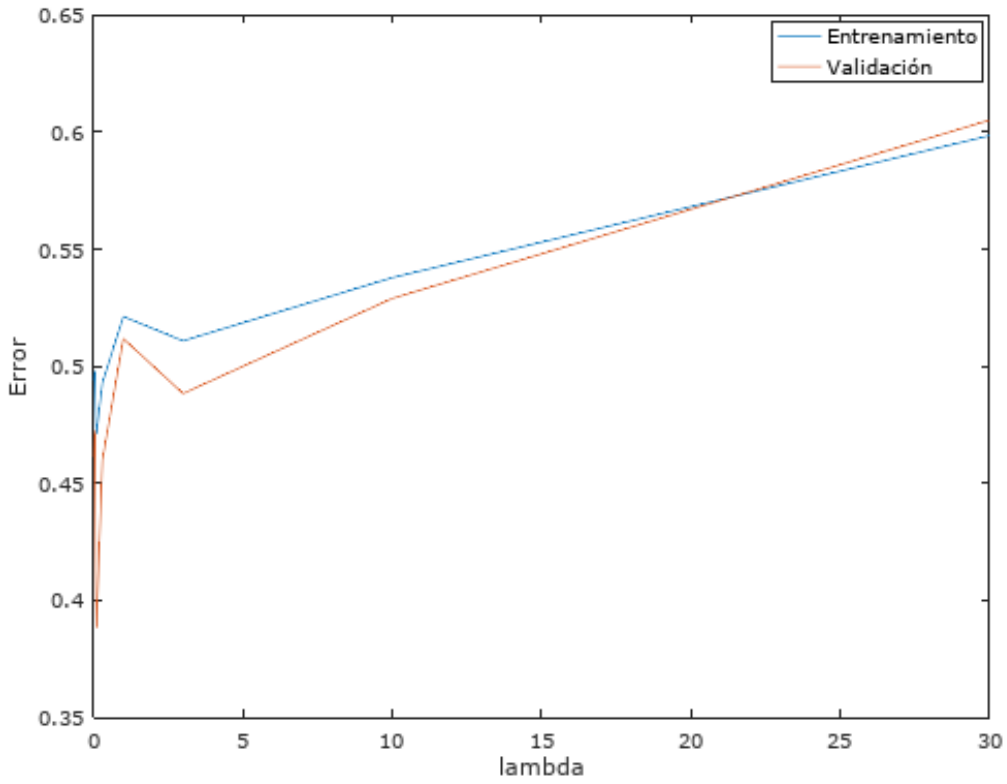


FIGURA 4.10: Curvas de evolución de los costes de entrenamiento y validación con un atributo adicional.

```
function [J grad] = costeRN(params_rn, num_entradas, num_ocultas,num_etiquetas,
                            X, y, lambda)
%reconstruir las matrices de parámetros a partir del vector params_rn
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));

% variables utililes
J = 0;
Theta1_grad = zeros(size(Theta1));
Theta2_grad = zeros(size(Theta2));
m = size(X, 1);

% para poder entrenar la red neuronal es necesario codificar las etiquetas como
%vectores de 10 componentes con todos sus elementos a 0 salvo uno a 1
I = eye(num_etiquetas);
Y = zeros(m, num_etiquetas);

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

Y=aux;
% feedforward
a1 = [ones(m, 1) X];
z2 = a1*Theta1';
a2 = [ones(size(z2, 1), 1) fsigmoide(z2)];
z3 = a2*Theta2';
a3 = fsigmoide(z3);
h = a3;
% calculo de la penalizacion
p = sum(sum(Theta1(:, 2:end).^2, 2))+sum(sum(Theta2(:, 2:end).^2, 2));
% calculo de J, observese que si lambda es 0 se devolvera sin el termino
%regularizado
J = sum(sum((-Y).*log(h) - (1-Y).*log(1-h), 2))/m + lambda*p/(2*m);

%RETROPROPAGACIÓN
% calculate sigmas
sigma3 = a3.-Y;
sigma2 = (sigma3*Theta2).*fsigmoideGradiente([ones(size(z2, 1), 1) z2]);
sigma2 = sigma2(:, 2:end);

delta_2 = (sigma3'*a2);
delta_1 = (sigma2'*a1);

%si lambda es 0 calculo se devolvera el gradiente sin regularizar ya que depende
%de lambda
p1 = (lambda/m)*[zeros(size(Theta1, 1), 1) Theta1(:, 2:end)];
p2 = (lambda/m)*[zeros(size(Theta2, 1), 1) Theta2(:, 2:end)];
Theta1_grad = delta_1./m + p1;
Theta2_grad = delta_2./m + p2;
% el gradiente se devuelve desplegado como un vector columna
grad = [Theta1_grad(:) ; Theta2_grad(:)];
endfunction
```

FIGURA 4.11: Código de la función costeRN

```
clear;
close all;
warning('off','all');
addpath("NN");
%cargamos los datos
data = csvread('data.csv');
%dividimos las features y los label y extraemos la feature ID(columna 2)
dta=data(1:end,:);
%no le añadimos aun los 1´s porque posteiormente se añadirán los bias unit
y= dta(:,1);
X= dta(:,3:end);

%posibles valores útiles
[m , n]=size(X);
%normalización
[X,mu,sigma] = normalizaAtributo(X);

%% ============= Inicializacion red Neuronal ==================================
%tamaño de las capas
input_layer_size  = 7;
hidden_layer1_size = 9;
output_layer_size = 1;

printf('====ENTRENAMIENTO DE LA RED NEURONAL DE UNA CAPA OCULTA DE 9N====\n\n');
printf('1-ENTRENAMIENTO DE LA RED NEURONAL CON EL TOTAL DEL DATASET\n \n');

%pesos iniciales aleatorios
initial_theta1 = pesosAleatorios(input_layer_size,hidden_layer1_size);
initial_theta2 = pesosAleatorios(hidden_layer1_size ,output_layer_size);

%enrrollado de parámetros
theta_inicial = [initial_theta1(:) ; initial_theta2(:)];

opciones = optimset('MaxIter', 100);
lambda = 3;

%chequeo retropropagación correcta con derivadas parciales
checkNNGradients(lambda);

%% ============= Entrenamiento red neuronal con el total de los datos ==========
tic;
params_rn = fmincg(@(t) (costeRN(t, input_layer_size, hidden_layer1_size,
              output_layer_size, X, y, lambda)), theta_inicial, opciones);
time = toc;
printf('->Se han obtenido los pesos optimos con un maximo de 100 iteraciones y
                              un parametro de regularizacion lambda = 3 \n');
[cost, grad] = costeRN(params_rn, input_layer_size, hidden_layer1_size,
                       output_layer_size, X, y, lambda);
printf('->El entrenamiento ha durado: %0.2f segundos y el coste minimo ha sido:
                                                          %f. \n', time, cost);
printf('Pulse Enter para continuar...\n \n');
pause();
%% ===============Accuracy con el total de los datos ============================
[accu] = accuracyNN(params_rn, X, y, input_layer_size, hidden_layer1_size,
                    output_layer_size);
printf('->Precision entrenando con el total del dataset: %f \n\n', accu * 100);
```

FIGURA 4.12: Código del flujo principal de la primera fase.

```
printf('2-ENTRENAMIENTO DE LA RED NEURONAL CON EL 70 por ciento del dataset\n');
dataTr = data(1:916,:);
dataTest = data(917:end,:);

y_tr = dataTr(:,1);
X_tr = dataTr(:,3:end);
y_test = dataTest(:,1);
X_test = dataTest(:,3:end);

%posibles valores utiles
[m_tr , n_tr]=size(X_tr);
[m_test , n_test]=size(X_test)
%Normalización de atributos
[X_tr,mu,sigma] = normalizaAtributo(X_tr);
[X_test,mu,sigma] = normalizaAtributo(X_test);

%pesos iniciales aleatorios
initial_theta1 = pesosAleatorios(input_layer_size,hidden_layer1_size);
initial_theta2 = pesosAleatorios(hidden_layer1_size ,output_layer_size);
%enrrollado de parámetros
theta_inicial = [initial_theta1(:) ; initial_theta2(:)];
opciones = optimset('MaxIter', 100);
lambda = 3;
%% ============= Entrenamiento red neuronal con el 70% de los datos ==========
tic;
params_rn = fmincg(@(t) (costeRN(t, input_layer_size, hidden_layer1_size,
              output_layer_size, X_tr, y_tr, lambda)), theta_inicial, opciones);
time = toc;
printf('->Se han obtenido los pesos optimos con un maximo de 100 iteraciones y
                                  un parametro de regularizacion lambda = 3 \n');
[cost, grad] = costeRN(params_rn, input_layer_size, hidden_layer1_size,
                                          output_layer_size, X_tr, y_tr, lambda);

printf('->El entrenamiento ha durado: %0.2f segundos y el coste minimo ha sido:
                                                              %f. \n', time, cost);
printf('Pulse Enter para continuar...\n \n');
pause();
[accu] =accuracyNN(params_rn, X_test,y_test,input_layer_size,hidden_layer1_size,
                                                                output_layer_size);
printf('->Precision entrenando con el 70 por ciento del dataset, sobre el 30 por
                        ciento restante destinado a test: %f \n\n', accu * 100);
```

FIGURA 4.13: Código del flujo principal de la segunda fase.

```
printf('2-ENTRENAMIENTO DE LA RED NEURONAL CON EL 70 por ciento del dataset\n');
dataTr = data(1:916,:);
dataTest = data(917:end,:);

y_tr = dataTr(:,1);
X_tr = dataTr(:,3:end);
y_test = dataTest(:,1);
dataTr = data(1:785,:);
dataCv = data(786:1047,:);
dataTest = data(1048:end,:);

y_tr = dataTr(:,1);
X_tr = dataTr(:,3:end);
y_cv = dataCv(:,1);
X_cv = dataCv(:,3:end);
y_test = dataTest(:,1);
X_test = dataTest(:,3:end);

%posibles valores utiles
[m_tr , n_tr]=size(X_tr);
[m_cv , n_cv]=size(X_cv);
[m_test , n_test]=size(X_test);
%crea un vector lambda
lambda_vec = [0.01, 0.03, 0.1, 0.3, 1, 3, 10 30]';
minCost = realmax;

%normalizacion de los atributos
[X_tr,mu,sigma] = normalizaAtributo(X_tr);
[X_cv,mu,sigma] = normalizaAtributo(X_cv);
[X_test,mu,sigma] = normalizaAtributo(X_test);

%pesos iniciales aleatorios
initial_theta1 = pesosAleatorios(input_layer_size,hidden_layer1_size);
initial_theta2 = pesosAleatorios(hidden_layer1_size ,output_layer_size);
%enrrollado de parámetros
theta_inicial = [initial_theta1(:) ; initial_theta2(:)];
opciones = optimset('MaxIter', 100);

for i = 1:rows(lambda_vec)
    printf('Entrenando con lambda: %d \n',lambda_vec(i));
    %entrenamiento
    params_rn = fmincg(@(t) (costeRN(t, input_layer_size, hidden_layer1_size,
       output_layer_size, X_tr, y_tr, lambda_vec(i))), theta_inicial, opciones);
    %guardamos los valores para posteriormente construir las gráficas
    [jtrain(i), grad] = costeRN(params_rn, input_layer_size, hidden_layer1_size,
                                output_layer_size, X_tr, y_tr, lambda_vec(i));
    [jval(i), grad] = costeRN(params_rn, input_layer_size, hidden_layer1_size,
                                output_layer_size, X_cv, y_cv, lambda_vec(i));

    if(jval(i) < minCost)
       minCost = jval(i);
       minCostCv = jtrain(i);
       bestlambda = lambda_vec(i);
     endif
endfor
printf('->Valor de lambda optimo: %f \n\n', bestlambda );
%entrenamiento con lambda elegido
params_rn = fmincg(@(t) (costeRN(t, input_layer_size, hidden_layer1_size,
          output_layer_size, X_tr, y_tr, bestlambda)), theta_inicial, opciones);
%porcentaje de aciertos en el test
[accu] = accuracyNN(params_rn,X_test,y_test,input_layer_size,hidden_layer1_size,
printf('->Precision sobre datos de test: %f \n\n', accu * 100);
```

FIGURA 4.14: Código del flujo principal de la tercera fase.

```
%=========================CURVA DE APRENDIZAJE=====================================
%recordar que para las curvas de aprendizaje es necesario que el parametro de
%Importante recordar regularización sea 0 para no incluirlo
lambda = 0;
for i = 1:m_tr
%entrenamos
 params_rn = fmincg(@(t) (costeRN(t, input_layer_size, hidden_layer1_size,
output_layer_size, X_tr(1:i,:), y_tr(1:i,:), lambda)), theta_inicial, opciones);
%almacenamos los costes
  [jtrain(i), grad] = costeRN(params_rn, input_layer_size, hidden_layer1_size,
                          output_layer_size, X_tr(1:i,:), y_tr(1:i,:), lambda);
  [jval(i), grad] = costeRN(params_rn, input_layer_size, hidden_layer1_size,
                                       output_layer_size, X_cv, y_cv, lambda);
endfor

%pintado de curva de aprendizaje
figure(2);
plot(1:m_tr, jtrain, 1:m_tr, jval);
title('Curva de aprendizaje para la red neuronal')
legend('Entrenamiento', 'Validacion')
xlabel('Numero de ejemplos de entrenamiento')
ylabel('Error')

printf('\n Pulse Entrer para finalizar.\n');
pause;
```

FIGURA 4.15: Código del flujo principal donde se construyen las curvas de aprendizaje.

Capítulo 5

Support Vector Machines

En este capítulo describiremos el algoritmo de support vector machines (SVM), que hemos aplicado e implementado para este proyecto, a nuestro dataset.

5.1. Normalización de los atributos

Antes de proceder con el algoritmo, hemos separado el *dataset* en atributos y resultado, y se han creado con esto las matrices correspondientes, tal y como hemos hecho con los algoritmos de regresión logística y la red neuronal

Igualmente, como pasaba con los dos algoritmos mencionados, decidimos no normalizar los atributos, puesto que, como ya hemos mencionado, los valores de estos se encuentran en rangos próximos y normalizar podría conducir a errores.

5.2. Entrenamiento y predicción

Al igual que hemos hecho con los dos algoritmos anteriores, hemos dividido en fases la implementación y aplicación de este algoritmo. A diferencia de cómo hemos atacado las fases en los algoritmos de regresión logística y redes neuronales, en este caso vamos a dividir nuestro dataset directamente en tres, una para entrenar (60 %), otra para validación (20 %) y otro para test (20 %); esto se debe a los altos tiempos de computo que va a requerir este algoritmo para un dataset del tamaño empleado. El flujo principal se encuentra en el fichero titanicSVM.m, adjunto con esta memoria. Las fases son las siguientes:

- Primera fase: tras separar los datos, hemos entrenado nuestra SVM, gracias a la función svmTrain, con un kernell lineal, cuyo código se adjunta en la figura 5.1. Para este entrenamiento hemos utilizado el 60 % del dataset, y el porcentaje de aciertos, calculado por la función svmPredict sobre los datos de test, ha sido de 100 %, y sobre los propios datos con los que hemos entrenado de 78.85 %. El tiempo para entrenar el modelo de esta manera ha ido de 542.84 segundos y el valor para el parámetro C ha sido de 1.

  El código asociado a esta parte del flujo principal es el que aparece en la figura 5.5.

```
function sim = linearKernel(x1, x2)
%LINEARKERNEL returns a linear kernel between x1 and x2
%   sim = linearKernel(x1, x2) returns a linear kernel between x1 and x2
%   and returns the value in sim

% Ensure that x1 and x2 are column vectors
x1 = x1(:); x2 = x2(:);

% Compute the kernel
sim = x1' * x2;  % dot product

end
```

FIGURA 5.1: Código de la función del kernell lineal.

- Segunda fase: en esta fase vamos a replicar la fase anterior pero utilizando en este caso un kernell gausiano, cuya especificación se aprecia en la figura 5.2. Los valores de C y sigma en este caso elegidos han sido de 1 y 0.1 respectivamente. El porcentaje de aciertos sobre los datos de test, tras el entrenamiento, ha sido de 62.21 %, y sobre los propios datos con los que hemos entrenado de 97.83 % Para entrenar nuestro modelo hemos necesitado un tiempo de 26.63 segundos.

```
%función que calcule el kernel gaussiano para así poder entrenar una SVM que
%clasifique correctamente el segundo conjunto de datos
function sim = gaussianKernel(x1, x2, sigma)
x1 = x1(:); x2 = x2(:);

%la función de kernel gaussiano calcula la distancia entre dos ejemplos de
%entrenamiento (x(i), x(j))
sim = 0;
sim = exp(-1*(x1-x2)'*(x1-x2)/(2*sigma*sigma));

endfunction
```

FIGURA 5.2: Código de la función del kernell gausiano.

El código asociado a esta parte del flujo principal es el que aparece en la figura 5.6.

- Tercera fase: en esta fase hemos decidido calcular los sigma y C óptimos para entrenar con nuestro kernell gausiano, para ellos e ha implementado la función que aparece en la figura 5.3. Tras este cálculo hemos obtenido unos valores para sigma y para C óptimos de 30 y 30 respectivamente. Con estos dos valores hemos vuelto a entrenar nuestro modelo de SVM y hemos calculado los aciertos sobre los datos de test, que en este caso han sido de 88,54 %.

  El código asociado a esta parte del flujo principal es el que aparece en la figura 5.7.

- Cuarta fase: por último, hemos desarrollado una función para calcular la precisión y el recall, como hemos hecho con los otros dos algoritmos. No hemos calculado en SVM las curvas de aprendizaje por el excesivo tiempo. El código de esta función se puede ver en la figura

```
function [C, sigma] = calculoCSigma(X, y, Xval, yval)
minerror = inf;
values = [0.01 0.03 0.1 0.3 1 3 10 30];
%doble for para los 64 modelos diferentes->8 valores de C por 8 valores de sigma
for Caux = values
  for sigmaAux = values
    printf('Entrenamiento y validacion para los valores
                          de: \n[Caux, sigmaAux] = [%f %f]\n', Caux, sigmaAux);

   %calculando el porcentaje de estos ejemplos que clasificaca correctamente
    model = svmTrain(X, y, Caux, @(x1, x2) gaussianKernel(x1, x2, sigmaAux));

   %calculo de los que se clasifican mal
    error = mean(double(svmPredict(model, Xval) ~= yval));
    printf('Prediccion de error: %f\n', error);

    %actualizacion del error minimo en caso de haber encontado uno menor
    if( error <= minerror )
      sigma = sigmaAux;
      C = Caux;
      minerror = error;
    endif

  endfor
endfor
  printf('Valor de C optima:  %f, valor de sigma optima: %f \n', C, sigma);
  printf('Error de prediccion minima de:  %f\n', minerror);
  printf('------------------------------------------------------------------\n');
endfunction
```

FIGURA 5.3: Código de la función que calcula los valores C y sigma óptimos.

5.4. El resultado tras utilizar kernell gausiano ha sido de 95.78 % de recall y 77.77 % de precisión.

```
function [precision, recall] = precisionrecall(ytest,p)
truepos = ytest + p == 2;
    if (p == 0)
        precision = 1;
    else
        precision = sum(truepos)/ sum(p);
    endif
    recall = sum(truepos) / sum(ytest);

    printf('Precision: %f,  Recall: %f \n', precision * 100, recall * 100);
endfunction
```

FIGURA 5.4: Código de la función que calcula el recall y la precisión.

```
%entrenamiento con kernell lineal
printf('=====FASE 1: SVM CON KERNELL LINEAL=====\n \n');
C = 1;
%====================================KERNELL LINEAL
tic;
model = svmTrain(X_tr, y_tr, C, @linearKernel, 1e-3, 20);
time = toc;
printf('->Se ha calculado el modelo con un maximo de 20 iteraciones y un valor
                                                                    de C = 1 \n')
printf('->Duracion del proceso %.2f segundos\n', time);

%calcula los aciertos con el total de los entrenados y un kernell lineal %
p = svmPredict(model,X_tr); %
printf('->Porcentaje de acierto con kernell linel entrenando y prediciendo con
            los datos de entrenamiento: %f\n', mean(double(p == y_tr)) * 100);
printf('Pulse Enter para continuar...\n');
pause;
%calcula los aciertos sobre los datos de test y un kernell lineal %
p = svmPredict(model,X_test); %
printf('->Porcentaje de acierto con kernell linel prediciendo sobre los datos de
                                    test: %f\n', mean(double(p == y_test)) * 100);
printf('Pulse Enter para continuar...\n');
pause;
```

FIGURA 5.5: Código asociado a la primera fase de SVM.

```
printf('=====FASE 2: ENTRENAMIENTO DEL SVM CON KERNELL GAUSIANO\n');
C = 1;
sigma = 0.1;

tic;
model = svmTrain(X_tr, y_tr, C, @(x1,x2) gaussianKernel(x1,x2,sigma));
time = toc;
printf('->Se ha calculado el modelo con un maximo de 20 iteraciones y un valor
                                                    de C = 1 y sigma=0.1 \n')
printf('->Duracion del proceso: %f segundos. \n', time);

%calculo del accuracy con el total entrenado
p = svmPredict(model, X_tr);
printf('Porcentaje de acierto del modelo sobre los datos de entrenamiento: %f\n'
                                        mean(double(p == y_tr)) * 100);
printf('Pulse Enter para continuar...\n');

%calculo del accuracy con el total entrenado
p = svmPredict(model, X_test);
printf('Porcentaje de acierto del modelo sobre los datos de test: %f\n',
                                      mean(double(p == y_test)) * 100);
printf('Pulse Enter para continuar...\n');
pause;
```

FIGURA 5.6: Código asociado a la segunda fase de SVM.

```
%entrenamiento con kernell gausiano
printf('=====FASE 4: ENTRENAMIENTO DEL SVM CON KERNELLGAUSIANO CSIG. OPTIMOS\n')
%Elección de los parámetros C y sigma optimos
[C, sigma] = calculoCSigma(X_tr, y_tr, X_cv, y_cv);

printf('->Valores para C y para sigma optimos son %f y %f.\n\n', C, sigma);
%entrenamos el modelo con estos valores devueltos en la funcion anterior
model= svmTrain(X_tr, y_tr, C, @(x1, x2) gaussianKernel(x1, x2, sigma));

p = svmPredict(model, X_test);
printf('Porcentaje de acierto del modelo entrenado con kernell gausiano y los
            valores C y sigma optimos: %f\n', mean(double(p == y_test)) * 100);
printf('Pulse Enter para continuar...\n');
pause;
```

FIGURA 5.7: Código asociado a la tercera fase de SVM.

Capítulo 6

Comparación de resultados

En este capítulo se exponen juntos los resultados obtenidos tras aplicar cada uno de los algoritmos de clasificación expuestos en los capítulos anteriores. Para facilitar el análisis de los resultados expuestos en el capítulo 7, estos se han ido facilitando tras aplicar cada fase de cada uno de los algoritmos desarrollados y expuestos en los capítulos previos, pero en esta tabla se expondrán los más relevantes y tras aplicar cross-validation y sobre ambos datasets. Estos resultados aparecen expuestos en la tabla 6.1.

	R.log 7atr.	R.log 8atr.	NN 7atr.	NN 8atr.	SVM(gauss)
Accuracy	96.56	92.74	74.80	63.74	88.54
Precision	97.80	89.79	25.19	36.25	77.77
Recall	93.68	92.63	100	100	95.78
Tiempo entrenamiento(seg)	0.08	0.10	0.49	0.56	26.63
Lambda óptimo	0.01	0.01	0.01	1	-
C óptimo -	-	-	-	-	30
Sigma óptimo -	-	-	-	-	30

TABLA 6.1: Resultado principales de los tres algoritmos de clasificación.

Capítulo 7

Conclusiones finales

En este capítulo vamos se exponen las conclusiones a las que hemos llegado, con los resultados obtenidos tras la complementación y ejecución de cada uno de los algoritmos desarrollados.

7.1. Conclusiones regresión logística.

Tras aplicar las diferentes fases en el algoritmo de regresión logística implementado para este proyecto, se ha llegado a las siguientes conclusiones:

- Tras la aplicación de cross-validation y mostrar las curvas de aprendizaje y error en función de lambda somos capaces de evaluar la calidad de nuestro clasificador.
- Si nos fijamos en la gráfica de las curvas de aprendizaje obtenidas tras aplicar cross-validation, la cercanía entre la curva de error de validación y la de error entrenamiento, podemos pensar que nos encontramos ante un problema de alto sesgo (high bias), el cual va asociado al underfitting, con lo que el aumento de datos de entrenamiento, aunque en nuestro caso no sería posible, no va a ayudar a paliar este problema.
- El bajo valor de lambda optimo también es síntoma de alto sesgo y por tanto de underfitting.
- Como ha quedado demostrado al reducirse el porcentaje de aciertos tras aumentar el número de atributos en uno, y habiendo identificado un posible caso de alto sesgo, no nos ha ayudado a mejorar nuestro modelo a pesar de lo esperado, con lo que es posible que el atributo elegido no aporte datos interesantes. Igualmente, los valores de precisión y recall se han visto mermados también al incrementar el dataset con este atributo.
- La selección de lambda, y la inclusión del término de regularización óptimo, calculado en base a los errores de validación, nos permite reducir el riesgo de overfitting.
- Para mejorar nuestro clasificador sería interesante plantear una hipótesis de mayor complejidad, es decir incrementar el grado del polinomio, e intentar de esta manera mejorar nuestro clasificador.
- También cabe destacar que el valor de la función de coste para el conjunto de datos de validación se reduce según vamos aumentando el número de ejemplos de entrenamiento. Lo

mismo ocurre con el error sobre los datos de entrenamiento, estabilizándose y manteniéndose muy cercanos ambos, a partir de los 100 ejemplos de entrenamiento aproximadamente. Por tanto, no sería una buena opción mejorar nuestro clasificador decidiendo incrementar el número de ejemplos de entrenamiento.

- Debido al alto porcentaje de aciertos alcanzado y el bajo coste computacional, el cual asumimos debido a los menores tiempos, y en comparación con las redes neuronales, consideramos este modelo una mejor opción frente a redes neuronales.

- Si nos fijamos en los valores de precisión y recall calculados, como una alternativa ante la posibilidad de encontrarnos a una clase sesgada, vemos que tenemos elevado valores para ambos, lo cual es deseable siempre.

7.2. Conclusiones redes neuronales.

Tras las fases que se describieron en el capítulo 4, y observando la tabla 6.1 expuesta en el capítulo 6, se han llegado a las siguientes conclusiones, con respecto al algoritmo de red neuronal implementado y aplicado en este proyecto:

- Observando las curvas de aprendizaje podemos notar que la red neuronal va disminuyendo considerablemente el error de validación, hasta estabilizarse cuando alcanzar aproximadamente los mismos ejemplos de entrenamiento que como pasaba en la regresión lineal, momento en el que comienza a tomar valores cercanos a los del error de entrenamiento, según sigue incrementándose el tamaño de los ejemplos de entrenamiento. Por tanto, incrementar el número de ejemplos de entrenamiento no ayudaría a mejorar nuestro clasificador.

- El coste de entrenamiento aumenta aproximadamente hasta alcanzar la misma cifra. Por tanto, alcanzada esta cifra de datos de entrenamiento no sería necesario seguir incrementando el tamaño del dataset para entrenar y validar nuestra red neuronal. Igual que en el caso de la regresión logística nos encontramos ante un caso posible de alto sesgo, pero como paso con esta, aumentar el número de atributos no nos ayudó.

- Nuevamente el hecho de incrementar las neuronas e incrementar el número de atributos, y por consiguiente el número de neuronas de la capa de entrada de la capa oculta no ha provocado un aumento en el número de aciertos, sino una disminución, tal como ocurrió en la regresión logística. La construcción de nuevos atributos ha supuesto el empeoramiento en el porcentaje de aciertos de la red, pasando de ser de un $75.57\,\%$ a un $63{,}74\,\%$, en ambos casos usando cross-validation.

- Frente a los algoritmos de regresión logística regularizada, no se ha alcanzado un mayor número de aciertos. Además, si añadimos el coste computacional más elevado de las redes neuronales, sobre todo si aumentamos las iteraciones y el número de neuronas de la capa oculta, podemos llegar a la conclusión de que no nos sería interesante desarrollar un modelo de clasificación binaria usando este algoritmo para este ejemplo.

- En cuanto a los valores de recall y precisión calculados, podemos apreciar que también se han visto mermados con respecto a los obtenidos en regresión lineal. Una precisión sumamente mala, es decir solo el 25.19 % de los que se han clasificado como ahogados se ahogaron realmente. Aunque el recall ha aumentado hasta 100 %, el valor de precisión hace pensar lo mal modelo de clasificación que es esta red neuronal. Al añadir un atributo más al dataset, a pesar de un menor número de aciertos (accuracy), se ha conseguido un mayor recall 36.25 % manteniendo la precision al 100 %, lo cual son un valor de recall más optimista y que indica que nuestro clasificador si ha mejorado un poco. Por todo esto no elegiríamos redes neuronales como algoritmo para crear un clasificador para nuestro ejemplo, a no ser que mediante la construcción de nuevos atributos y eliminación de otros innecesarios consiguiéramos aumentar mucho las métricas de evaluación utilizadas, es decir accuracy, recall y precision.

7.3. Conclusiones SVM.

Tras implementar nuestra SVM para este proyecto, se ha llegado a las siguientes conclusiones:

- El tiempo de entrenamiento de la SVM con kernell gausiano es mucho menor que con kernell lineal ara mismos valores de parámetro C.
- Los resultados en comparación con la red neuronal, son notoriamente mejores.
- Si comparamos con la regresión logística, en tiempo de cálculo se empeora, tanto con kernell gausiano como con kernell lineal.
- La precisión y el recall, son mejores también que en redes neuronales. Si los comparamos con los valores obtenidos en regresión logística sin embargo no consiguen igualarlos en cuanto a precision, sin embargo, en recall lo supera en un 2 %. Por lo que sumado a sus altos tiempos de computo, lo encontramos en desventaja frente al modelo desarrollado de regresión logística.