APRENDIZAJE AUTOMÁTICO Y BIG DATA

DETECCIÓN DE TUMORES CEREBRALES

PRÁCTICA FINAL

RAUL DIEGO NAVARRO

ÍNDICE:

1. INTRODUCCIÓN

En este estudio voy a poner en práctica los conocimientos adquiridos en la asignatura de aprendizaje automático y Big Data. Para ello he elegido un *dataset* que contenga un conjunto de datos que me permita tratarlos con cada método estudiado en la asignatura, para obtener una conclusión correcta en cada uno de ellos. Este *dataset* se puede encontrar en el siguiente enlace:

https://www.kaggle.com/jakeshbohaju/brain-tumor

Con este conjunto de datos voy a realizar una investigación que me permita determinar si un paciente al que se le han hecho determinadas pruebas tiene o no un tumor cerebral. El resultado de estas pruebas son imágenes que nos proporcionan los datos que encontramos en el *dataset*.

En él se describen en 17 columnas de atributos que lo forman. Estos atributos están clasificados en tres grupos:

- Características de primer orden
    - Media
    - Diferencia
    - Desviación Estándar
    - Oblicuidad
    - Curtosis
- Características de segundo orden
    - Contraste
    - Energía
    - ASM (segundo momento angular)
    - Entropía
    - Homogeneidad
    - Disimilitud
    - Correlación
    - Tosquedad
- Parámetro de evaluación de características
    - PSNR (relación pico señal-ruido)
    - SSIM (Índice de similitud estructurada)
    - MSE (error cuadrático medio)

- DC (coeficiente de dados)

Cabe destacar que el *dataset* también tiene una columna que está formada por la cadena 'ImageX', siendo X el numero de la fila, es decir, un ejemplo de entrenamiento de un paciente anterior, y otra columna final donde nos exponen si la imagen tiene un tumor o no (1 = tumor; 0 = no tumor).

De todos estos atributos he hecho una preselección, descartando todos aquellos que tenían algunas posiciones vacías (sí es verdad, que podría rellenarlos con la media, la moda, un cero o un uno, dependiendo del tipo de dato, pero en este caso al contar con varias columnas creo que me es suficiente), o aquellos que no me ayudan en el estudio que llevo a cabo en esta práctica, como puede ser "Tosquedad", que todos los datos de esa columna son iguales, por tanto, no tienen relevancia en este estudio. Una vez realizada esta preselección he pasado de los 17 a los 11 atributos.

Ya sabemos que partimos de:

- n = [1,11], siendo "n" el número de atributos.
- m = 1644, siendo "m" el número de ejemplos de entrenamiento que posee el *dataset*.

Siguiendo la condición vista en clase:

- Si "n" es pequeña y "m" intermedia => Usamos el método de vectores SVM con el *kernel gaussiano.*

Y aunque esto sería lo más adecuado, voy a estudiar los siguientes tres métodos: regresión logística, redes neuronales y vectores SVM. He dividido los métodos prácticamente en los mismos apartados para que el estudio sea uniforme y se pueda llegar a una conclusión de una manera más sencilla.

También es preciso destacar que la plataforma elegida para el estudio es "Jupyter Notebook", ya que es la que más he utilizado en las prácticas. Por tanto, parte de este .PDF contendrá capturas de la implementación en esta plataforma.

Por último, señalar que las librerías usadas en este proyecto son las mismas que las usadas en las prácticas. Si en algún caso se hace uso de alguna que no haya sido utilizada en las mismas, se especificará.

2. REGRESIÓN LOGÍSTICA

Empiezo el estudio, con la regresión logística, para seguir el orden cronológico llevado en clase. El objetivo de este método es estimar la probabilidad de un que un paciente tenga un tumor cerebral en base a los atributos obtenidos del *dataset*.

2.1. Elección de los atributos

Este apartado, aunque está situado el primero en cada categoría no se va a cerrar hasta el final de la implementación de cada método. Debido a que creo que los correcto es probar el algoritmo final con conjuntos distintos de atributos.

Está el primero con la intención de exponer las distintas formas con las que voy a tratar estos atributos. El objetivo final de esta elección es seleccionar de entre estos 11 atributos que tenemos después de la preselección sólo los dos que mejor resultado den (mayor porcentaje de acierto en las predicciones), teniendo en cuenta que se consiga un nivel de error "asumible". Si no se consiguiera esto, pasaría a tres y así sucesivamente. También voy a estudiar los datos según la categoría de estos, es decir, estudiaré el algoritmo usando sólo los atributos de cada orden, juntando un atributo de cada orden, etc.

En el apartado de análisis de los resultados se expondrá la conclusión con respecto al número y que atributos tienen el mayor porcentaje de acierto.

2.2. Implementación de las funciones necesarias

La primera función necesaria es la de carga de datos en las variables correspondientes, común a todos los métodos y que, por tanto, sólo se expondrá una vez.

```
def carga_csv(file_name):
    valores = read_csv(file_name, header=None).values
    return valores
```

Esta función es la misma que usamos en la práctica 2 pero con la diferencia que no convierte los tipos en "*float*", ya que recordemos que la primera columna está formada por "*strings*".

El segundo método que voy a exponer también es común a todos los métodos y solo se expondrá una vez.

```
def  seleccionAtributos(datos):
    datos = datos[:,1:].astype(float)
    datos = datos[:, (11,10,15,17)]
    return datos
```

Con esta función lo que se pretende es primero seleccionar todas las columnas menos la primera y convertirlos en *float*. La segunda instrucción establece una forma rápida de elegir los atributos que quiero evaluar (En el ejemplo los atributos seleccionados son 10, 11, 15, además del 17 que es el de clasificación).

Seguimos con funciones más específicas de este método:

```
def funcion_sigmoide(z):
    return (1/(1 + np.exp(-z)))
```

La función *funcion_sigmoide* es capaz de calcular la función sigmoide de un número, vector o matriz. Este método permite describir una progresión temporal desde unos niveles bajos al inicio, hasta acercarse a un clímax transcurrido un cierto tiempo.

```
def h(X, theta):
    return funcion_sigmoide(X.dot(theta))
```

La función *h* se encarga de calcular la hipótesis recibiendo como parámetros dos matrices X y theta, y haciendo una llamada al método anterior.

```
def costeReg (theta, X, y, lamda = 1):
    J = -1/len(X) * ( ( (np.log( h(X ,theta)  ) ).T).dot( y )  +
                ( (np.log( 1 - h(X, theta)  ) ).T).dot( 1 – y ) +
                (lamda/( 2 * len( X ) ) * (np.sum( theta ) )**2) )
    return J
```

La función *costeReg* calcula la función de coste, que en regresión logística viene dada por la expresión:

$$J(\theta) = -\frac{1}{m}((log\,(g(X\theta)))^T y + (log\,(1 - g(X\theta)))^T(1 - y)) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

```
def gradienteReg(theta, X, y, lamda = 1):
    grad = 1/len(X) * (X.T).dot(h(X, theta) - y)
    grad[1:] += theta[1:] *(lamda/len(X))
    return grad
```

La función *gradienteReg* calcula el gradiente de la función, que en regresión logística viene dada por la expresión:

$$\frac{\delta J(\theta)}{\delta \theta_j} = \frac{1}{m}X^T(g(X\theta) - y) + \frac{\lambda}{m}\theta_j$$

```
def mapFeature (X, p):
    poly = skp.PolynomialFeatures(p,include_bias=False)
    return poly.fit_transform(X)
```

La función *mapFeature* se encarga de transformar la matriz X en un polinomio de grado "p", es una versión de la usada en la práctica 2 y 5, por tanto, no hay mucho más que explicar.

```
def  normalizar (X):
   mu = np.mean(X,0)
   s = np.std(X,0)
   X = ( X – mu )/s
   return ( X, mu, s )
```

La función *normalizar* se encarga de la normalización de los atributos debido a que no queremos que se introduzcan grandes diferencias de rango. Este método hace uso de funciones que calculan la media aritmética y la desviación estándar.

```
def  grafica_Positivos_Negativos (X, y):
    plt.figure()

    pos = (y == 1).ravel()
    neg = (y == 0).ravel()

   plt.scatter(X[pos, 0], X[pos, 1], color='black', marker='+')
   plt.scatter(X[neg, 0], X[neg, 1], color='yellow', edgecolors='black',
marker='o')
    return
```

La función *grafica_Positivos_Negativos* se encarga de elaborar la gráfica que simula los datos diferenciándolos, dependiendo de si "y = 0" o "y = 1".

```
def  porcentajeAciertos (X, Y, theta):
     numFilas = len(X)
     aciertos = 0
     for i in range(0,numFilas):
         xx = np.transpose(X[i,:])
         xx = np.dot(xx, theta)
         hip = funcion_sigmoide(xx)
         if(hip < 0.5):
             hip = 0
         else:
             hip = 1
         if(Y[i] == hip):
             aciertos += 1
       return (aciertos/numFilas)*100
```

Por último, en este apartado, la función *porcentajeAciertos* calcula el porcentaje de aciertos del algoritmo. Esta función comprueba el resultado de la hipótesis que ha calculado utilizando los datos elegios y el vector de theta optimizado con los ejemplos de entrenamiento.

Una vez que ya he expuesto todas las funciones necesarias para implementar este algoritmo, cito los pasos que he seguido para llevar a cabo los cálculos de la regresión logística. Estos pasos sólo los voy a exponer una vez, ya que me parece inútil repetir todos los pasos de todos los casos que voy a realizar, eso sí, en el análisis de los resultados, profundizaré sobre la mayoría de las pruebas realizadas.

Los atributos elegidos para este ejemplo son MSE (parámetro de evaluación) y Correlation (segundo orden). He elegido estos parámetros debido a que para los cálculos con sólo dos atributos son los que más porcentaje de aciertos han obtenido.

Comenzamos con la obtención de los datos requeridos, primero llamando a la función de carga y luego a la de selección. Posteriormente introducimos estos datos en las variables *X* e *y*, y comprobamos su forma para cerciorarnos de que se ha ejecutado correctamente.

```
datos = carga_csv('bt_dataset_t3.csv')

datos = seleccionAtributos(datos)
X = datos[:,:-1]
y = datos[:, -1]
print(np.shape(X))
print(np.shape(y))
```

```
(1644, 2)
(1644,)
```

Una vez vemos que las formas coinciden con los valores esperados, procedemos a inicializar las variables que vamos a usar para dividir los datos: entrenamiento, validación y test. Como hemos comprobado el valor de n es 1644 y puesto que he elegido 60%, 20% y 20% respectivamente, el código quedará así:

```
#60%
numTrain = 987
#20%
numVal = 328
#20%
numTest = 329
```

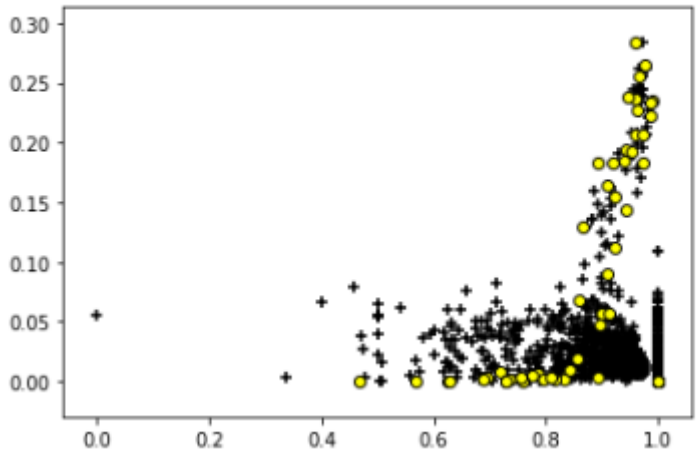
El siguiente paso es dividir los datos usando estas variables enteras.

```
xtrain = X[:numTrain,:]
ytrain = y[:numTrain]
print(np.shape(xtrain))
print(np.shape(ytrain))

xval = X[:numVal,:]
yval = y[:numVal]
print(np.shape(xval))
print(np.shape(yval))

xtest = X[:numTest,:]
ytest = y[:numTest]
print(np.shape(xtest))
print(np.shape(ytest))
```

```
(987, 2)
(987,)
(328, 2)
(328,)
(329, 2)
(329,)
```

Como podemos ver, los datos han sido divididos correctamente y ya podemos empezar con los cálculos pertinentes, no sin antes, observar el gráfico de los datos seleccionados.

```
grafica_Positivos_Negativos(xtrain, ytrain)
```



Para empezar los cálculos, vamos a transformar los datos en un polinomio de grado 12, normalizar estos datos y comprobar su forma.

```
x = mapFeature(xtrain, 12)
(x, mus , s) = normalizar(x)
print(x.shape)
```

```
(987, 90)
```

Sigamos con la introducción de 1's en la matriz x, la inicialización de los pesos con el tamaño de fila de *x*, cálculo del coste regularizado y el gradiente de la función.

```
x = np.hstack([ np.ones((len(x),1)), x])
print(x.shape)
theta = np.zeros((x[0].size))
print(np.shape(theta))
```

```
(987, 91)
(91,)
```

```
print(costeReg(theta, x,ytrain, 1))
print(gradienteReg(theta, x,ytrain, 1))
```

```
0.6931471805599453
[-0.38956434  0.03435386  0.02196256  0.03917981  0.02517136  0.05178793
  0.04336237  0.02718751  0.05199999  0.05324041  0.04715803  0.02852756
  0.05197442  0.05292571  0.05057307  0.0506038   0.02941219  0.05182353
  0.05257141  0.05021467  0.04738697  0.05372407  0.02997381  0.05159253
  0.05220184  0.04985788  0.04707619  0.04436044  0.05654364  0.03029694
  0.0513084   0.05182676  0.04950812  0.04677458  0.04410523  0.04162493
  0.05908788  0.03043835  0.05098832  0.0514519   0.04916739  0.04648311
  0.04385894  0.04141504  0.039186    0.06138181  0.0304378   0.05064383
  0.05108079  0.04883669  0.04620194  0.04362155  0.04121231  0.0390091
  0.0370186   0.06344935  0.03032418  0.05028295  0.05071567  0.04851646
  0.045931    0.04339285  0.04101652  0.03883757  0.03686422  0.03509247
  0.06531284  0.03011904  0.04991145  0.05035798  0.04820689  0.04567007
  0.04317256  0.04082745  0.03867122  0.03671372  0.03495264  0.03337915
  0.06699281  0.02983886  0.04953354  0.05000863  0.04790793  0.04541888
  0.04296039  0.04064485  0.03850987  0.03656699  0.03481554  0.03324815
  0.03185356]
```

Una vez que hemos comprobado que el gradiente y el coste no tienen errores de implementación, introducimos la función *opt.fmin_tnc* que nos ayuda a obtener el valor óptimo de *theta* para la versión regularizada del coste.

```
result = opt.fmin_tnc(func=costeReg, x0=theta, fprime=gradienteReg, args=(x,ytrain,1))
theta_opt = result[0]
```

Por último, calculamos el porcentaje de acierto del algoritmo con *theta* ya optimizada y lo imprimimos por consola.

```
percent = porcentajeAciertos(x,ytrain,theta_opt)
print("Porcentaje de aciertos es ", percent, "%")
```

```
Porcentaje de aciertos es  88.9564336372847 %
```

Podemos afirmar que el porcentaje de acierto no es todo lo alto que quisiéramos o al que podemos optar con este algoritmo, esto es por la selección de atributos que hemos hecho para esta prueba. Como hemos explicado anteriormente en los siguientes apartados, se profundizará en todas las pruebas realizadas y sus resultados.

2.3. Elección del parámetro lambda

En este apartado, voy a enseñar el código y las funciones utilizadas para elegir el valor de lambda más adecuado, es decir, aquel que obtienen un mayor número de aciertos.

Empecemos con las funciones implementadas en este apartado.

```
def regresionLogisticaReg (theta, X, y, lamb):
    y = y.ravel()
    return (costeReg(theta,X,y,lamb), gradienteReg(theta,X,y,lamb))
```

La función *regresionLogisticaReg* se encarga de devolver una tupla formada por el coste regularizado y el vector gradiente de la función.

```
def curvaAprendizaje (x, y, xval, yval, reg):
    trainError = np.zeros((len(x), 1))
    validationError = np.zeros((len(x), 1))

    for i in range(1, len(x) + 1):
        theta = np.zeros((x[0].size))
        thetaOpts = opt.minimize(regresionLogisticaReg, theta, args = (x[0:i],
y[0:i], reg), method='TNC', jac=True)
        thetaOpts = thetaOpts['x']

        trainError[i-1] = regresionLogisticaReg(thetaOpts, x[0:i], y[0:i], 0)[0]

        validationError[i-1] = regresionLogisticaReg(thetaOpts, xval, yval, 0)[0]

    return (trainError, validationError)
```

La función *curvaAprendizaje* repite el entrenamiento por regresión logística, utilizando diferentes subconjuntos de los datos de entrenamiento. También tenemos que evaluar el error del resultado, sobre cada subconjunto, así como el error al clasificar a todos los ejemplos del conjunto de validación.

```
def graficaCurvaAprendizaje (x, y, xval, yval, lamb):
    train, crossValidation = curvaAprendizaje(x, y, xval, yval, lamb)
    plt.plot(np.arange(1, len(x) + 1), train, 'b', label = 'train')
    plt.plot(np.arange(1, len(x) + 1), crossValidation, 'tab:orange', label =
'Cross Validation')
    plt.xlabel('Number of training examples')
    plt.ylabel('Error')
    plt.legend(loc=0)
```

La función *graficaCurvaAprendizaje* dibuja la gráfica elaborada con el método anteriormente expuesto.

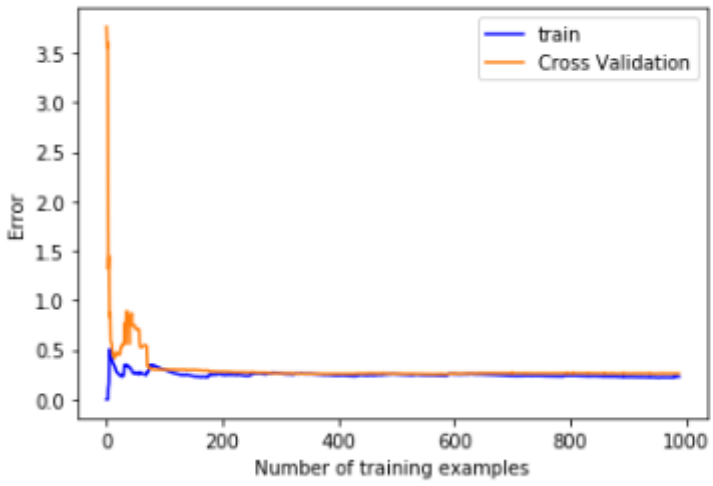
Paso a mostrar las capturas de las instrucciones que hay que introducir que nos llevan a la correcta ejecución de esta parte.

Empezamos con el manejo de los datos dedicados a la parte de validación con el objetivo de que tengan las dimensiones adecuadas.

```
aux = mapFeature(xval, 12)
polinomio_Xval = (aux-mus)/s
polinomio_Xval = np.hstack([ np.ones((len(polinomio_Xval),1)), polinomio_Xval])
```

Posteriormente, realizamos la llamada a la función *graficaCurvaAprendizaje* que se encargará de mostrarnos el gráfico que buscamos.

```
graficaCurvaAprendizaje(x, ytrain, polinomio_Xval, yval, 1)
```



Estos son los pasos que he realizado con cada uno de los conjuntos de datos que he seleccionado y que voy a exponer en el último apartado de esta sección. Además, debemos realizar varias llamadas con diferentes valores de lambda con el fin de obtener el valor más adecuado.

2.4. Prueba del algoritmo con los test

En este apartado se va a definir el código necesario para comprobar los resultados obtenidos con la parte de los datos dedicados a test.

Este código se resume en aplicar algunos de los pasos que hemos realizado hasta ahora, pero con los datos de los test. Lo primero que haríamos sería darle la forma adecuada, luego normalizaríamos los datos con los valores obtenidos anteriormente. Realizamos la optimización y obtenemos el error sobre los test para esos valores optimizados.

```
aux = mapFeature(xtest, 12)
polinomio_Xtest = (aux-mus)/s
polinomio_Xtest = np.hstack([ np.ones((len(polinomio_Xtest),1)), polinomio_Xtest])
theta = np.zeros((x[0].size))
thetaOpts = opt.minimize(regresionLogisticaReg, theta, args = (x, ytrain, 0.01),
                         method='TNC', jac=True)['x']
trainError = regresionLogisticaReg(thetaOpts, polinomio_Xtest, ytest, 0)[0]
print("Error obtenido con Xtest = ", trainError)
```

```
Error obtenido con Xtest =  0.2131065556553645
```

Como observamos en este ejemplo el error es bastante bajo, por lo que podremos decir que es un error “asumible”.

2.5. Análisis de los resultados

En este apartado voy a analizar todos los resultados obtenidos, con sus respectivos razonamientos. Voy a empezar con el ejemplo más obvio que es aquel con todos los atributos que preseleccionamos al principio de esta práctica.

Once atributos seleccionados

El valor elegido para el polinomio en este caso es 6 ya que creo suficiente para tal longitud de atributos. Lo primero es comprobar el porcentaje de aciertos que para este caso ha sido:

```
Porcentaje de aciertos es  95.23809523809523 %
```

Como vemos es un porcentaje bastante mejor que el obtenido en el ejemplo anterior. Sin embargo, creo excesivo la carga computacional a la que se somete a la máquina local, en los cálculos ejecutados.

Lo siguiente sería escoger el valor de lambda más adecuado. Según las pruebas que he realizado, los diferentes valores de lambda en el rango [0.5, 3] nos llevaría al valor más adecuado, ya que la diferencia de solución entre estos es insignificante.

Por último, el valor del error obtenido en los test es:

```
Error obtenido con Xtest =  0.16952511484216995
```

Que como podemos observar es bajo.

Selección por tipo de atributo

Como especifiqué en la introducción, seleccionar los atributos según la clasificación que ocupen entre los tres grandes grupos sería una buena manera de observar la importancia de cada uno de estos.

PRIMER ORDEN

Empezamos con los atributos de primer orden, que en este caso son tres, Media, Diferencia, Desviación Estándar.

El porcentaje de aciertos obtenidos con un polinomio de grado 8 es:

```
Porcentaje de aciertos es  89.05775075987842 %
```

El resultado de los test con lambda igual a 1 es:

```
Error obtenido con Xtest =  0.33015347820901403
```

Podemos observar que el error obtenido es mayor que el del ejemplo con solo dos atributos. Esto nos ayuda a hacernos una idea de que este grupo es menos importante que los otros dos.

SEGUNDO ORDEN

En este caso los atributos son:

- Contraste
- Energía
- ASM (segundo momento angular)
- Entropía
- Homogeneidad
- Disimilitud
- Correlación

El porcentaje de aciertos obtenidos con un polinomio de grado 8 es:

```
Porcentaje de aciertos es  90.17223910840933 %
```

El resultado de los test con lambda igual a 1 es:

```
Error obtenido con Xtest =  0.2765390049571202
```

En comparación con el grupo anterior el porcentaje es mejor, pero puede deberse al número de atributos. Ya que en el último grupo sólo nos queda un atributo no lo voy a exponer, pero sí que podemos llegar a la conclusión de que debe estar en el conjunto de atributos finalmente seleccionados, ya que el porcentaje del ejemplo, donde si estaba este último atributo, es ligeramente mejor.

Conjunto de tres atributos

En esta sección voy a exponer que tres atributos obtienen mejores predicciones. El objetivo es alcanzar un porcentaje de aciertos alto, con una carga computacional menor.

Después de varias pruebas, he llegado a la conclusión de que el conjunto de atributos que mejor funciona con respecto a consumo computacional y porcentaje de aciertos es: MSE, Correlación y Disimilitud.

El porcentaje de aciertos obtenidos con un polinomio de grado 12 es:

```
Porcentaje de aciertos es  95.74468085106383 %
```

El resultado de los test con lambda igual a 1 es:

```
Error obtenido con Xtest =  0.2253822025335773
```

En este caso tanto el porcentaje de aciertos como el error obtenido son los mejores resultados que he conseguido obtener. Por tanto, podemos afirmar que estos tres atributos serían los idóneos para predecir los tumores cerebrales con regresión logística.

3. REDES NEURONALES

Continuamos el estudio, con las redes neuronales. El objetivo de este método es clasificar entre tumor o no tumor en base a los atributos obtenidos del *dataset*. En este caso las redes neuronales no estiman una probabilidad de tener tumor cerebral, sino que directamente lo clasifican.

3.1. Elección de los atributos

Para este apartado de redes neuronales, voy a elegir todos los atributos que quedaron después de la preselección, ya que la estructura de la red depende del número de éstos y me parece la elección más adecuada. No obstante, en el último apartado haré referencia a los resultados obtenidos y a la eficacia de esta selección.

Pasamos a la parte de la implementación, recordar que no voy a hacer referencia a las funciones de carga y selección ya que son las mismas que en el apartado anterior.

3.2. Implementación de las funciones necesarias

Voy a introducir las funciones implementadas para este método, en orden incremental, puesto que hay más llamadas entre funciones que en los demás métodos y no quiero que haya confusión.

```
def despliegaMatriz(params_rn, num_entradas, num_ocultas, num_etiquetas):
    theta1 = np.reshape ( params_rn [:num_ocultas * (num_entradas + 1)],
  (num_ocultas, (num_entradas + 1)) )

  theta2 = np.reshape ( params_rn [num_ocultas * (num_entradas + 1):],
  (num_etiquetas, (num_ocultas + 1)) )

  return (theta1, theta2)
```

La función *despliegaMatriz* es la que se encarga de ejecutar el código dado en la práctica 4 y que forma la estructura de la red neuronal. He introducido este código en una función ya que consideraba importante que no se repitiera código en los siguientes métodos y quedará el código más legible y limpio.

```
def funcion_sigmoide(z):
    return  (1/(1 + np.exp(-z)))
```

Esta función es la misma que he utilizado en el algoritmo de regresión logística y, por tanto, no voy a profundizar más.

```
def hipotesis(theta1, theta2, x):
    one = np.ones((len(x), 1))
    a1 = np.hstack([one, x])
    a2 = funcion_sigmoide(a1.dot(theta1.T))
    a2 = np.hstack([one, a2])
    a3 = funcion_sigmoide(a2.dot(theta2.T))
    return a3
```

La función *hipotesis*, que es diferente a la implementada en el algoritmo anterior, se encarga de calcular la hipótesis en cada una de las capas de la red neuronal, y devolver el valor de esta en la última capa.

```
def funcionCoste(theta1, theta2, x, y, num_etiquetas):

    yk = np.zeros((len(x), num_etiquetas))
    for k in range(1, num_etiquetas + 1):
        y_ = np.where(y == k, 1, 0)
        yk[:, k-1] = y_.ravel()

    return -1/len(x) * np.sum((np.log( hipotesis(theta1, theta2, x) ) * yk)
```

Igual que en la regresión, la función *funcionCoste* calcula el coste del algoritmo que estamos valorando.

```
def funcionCosteReg (theta1, theta2, x, y, num_etiquetas, lamb):

    coste = funcionCoste(theta1, theta2, x, y, num_etiquetas)

    return coste + (lamb/(2*len(x)))*(np.sum(theta1[:,1:]**2)
    + np.sum(theta2[:, 1:]**2))    + (np.log( 1 - hipotesis(theta1, theta2, x) ) * (1-
yk)))
```

Esta función añade, a la de coste, la parte regularizada de tal manera que obtenemos el cálculo de la fórmula de coste regularizado en su totalidad.

```
def derivada_fun_sigmoide (z):

        return funcion_sigmoide(z) * (1 - funcion_sigmoide(z))
```

La función *derivada_fun_sigmoide* realiza el cálculo de la derivada de la función sigmoide que sigue la siguiente fórmula:

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1 - g(z))$$

```
def  pesosAleatorios (L_in, L_out):

        return np.random.uniform(-0.12, 0.12, (L_in, L_out))
```

La función *pesosAleatorios* es la misma que expone el .PDF de la práctica 4, y siguiendo los valores aconsejados en el rango [-0.12 , 0.12], que inicializa la matriz de pesos con otros valores aleatorios en este rango.

```
def  salidasCapas (theta1, theta2, x):
    a1 = np.append(1, x)
    z2 = a1.dot(theta1.T)
    a2 = funcion_sigmoide(z2)
    a2 = np.append(1, a2)
    z3 = a2.dot(theta2.T)
    a3 = funcion_sigmoide(z3)

    return (a1,a2,a3,z2)
```

La función *salidasCapas* calcula y devuelve cada una de las salidas de las capas además de la z que utilizamos para el cálculo en la segunda capa, ya que vamos a necesitarlo en la función que calcula el gradiente.

```
def  gradiente (x, y, theta1, theta2, num_etiquetas):
     delta1 = np.zeros((len(theta1), theta1[0].size))
     delta2 = np.zeros((len(theta2), theta2[0].size))


      yk = np.zeros((len(x), num_etiquetas))
     for k in range(1, num_etiquetas + 1):
        yaux = np.where(y == k, 1, 0)
        yk[:, k-1] = yaux.ravel()

   for i in range(0, len(x)):
     a1, a2, a3, z2 = salidaCapas(theta1, theta2, x[i,:])

     d3 = (a3 - yk[i])

     z2 = np.append(1, z2)
     d2 = (theta2.T).dot(d3)*dg(z2)

     d2 = np.delete(d2, 0)

     delta1 += np.matmul(d2[:,np.newaxis],a1[np.newaxis,:])
     delta2 += np.matmul(d3[:,np.newaxis],a2[np.newaxis,:])

   D1 = delta1/len(x)
   D2 = delta2/len(x)

   return np.concatenate((D1.ravel(), D2.ravel()))
```

La función *gradiente* calcula el vector de pesos de la red neuronal, como se puede observar, hago una llamada a la función anterior y calculo las delta requeridas.

```
def costeRN (params_rn, num_entradas, num_ocultas, num_etiquetas, X, y,
reg):
    theta1, theta2 = despliega(params_rn, num_entradas, num_ocultas,
num_etiquetas)

    coste = funCosteReg(theta1, theta2, X, y, num_etiquetas, reg)
  gradient = gradiente(X, y, theta1, theta2, num_etiquetas)

    delta1, delta2 = despliega(gradient, num_entradas, num_ocultas,
num_etiquetas)

    delta1[:, 1:] += (theta1[:, 1:] * reg/len(X))
    delta2[:, 1:] += (theta2[:, 1:] * reg/len(X))

    gradient = np.concatenate((delta1.ravel(), delta2.ravel()))

    return (coste,gradient)
```

Para terminar con este apartado, la función *costeRN* devuelve una tupla formada por el coste y gradiente regularizados.

3.3. Aprendizaje de la red neuronal

Debido a que no tengo los pesos adecuados para esta estructura y los datos con los que estoy llevando a cabo estas pruebas, no puedo verificar la correcta implementación de estas funciones, pero debido a que son los mismos que utilicé en la práctica 4 de esta asignatura, lo doy por comprobado.

A continuación, voy a exponer la función que utilizo para el aprendizaje de la red neuronal:

```
def aprendizaje (lamb, iteraciones):
    thetaRnd1 = pesosAleatorios(12, 4)
    thetaRnd2 = pesosAleatorios(5, 2)
    thetaRnd = np.r_[thetaRnd1.ravel(), thetaRnd2.ravel()]
    fmin = opt.minimize(costeRN, thetaRnd, args=(11,4,2,X,y, lamb),
                        method='TNC', options={'maxiter': iteraciones}, jac=True)

    optTheta = fmin.x
    theta1, theta2 = despliegaMatriz(optTheta, 11, 4, 2)
    probabilidad = hipotesis(theta1, theta2, X)
    indices  = np.argmax(probabilidad, axis=1)
    indices  += 1
    porcentaje = (indices == y.ravel())
    clasificados = porcentaje[porcentaje == True]
    percent = (len(clasificados)/len(X))*100

    print("Porcentaje de aciertos: ", percent, "%")
```

Esta función entrena la red neuronal y devuelve el porcentaje de aciertos después de la optimización de los valores de theta.

3.4. Elección de los parámetros lambda e iteración

Una vez expuestas todas las funciones de necesarias para la correcta ejecución de este algoritmo, quedaría elegir que valores de lambda e iteración obtienen el mejor resultado.

Antes de esto voy a enunciar las instrucciones seguidas para extraer y seleccionar los datos correctamente:

```
datos = carga_csv('bt_dataset_t3.csv')

datos = seleccionAtributos(datos)
X = datos[:,:-1]
y = datos[:, -1]
print(np.shape(X))
print(np.shape(y))
```

```
(1644, 11)
(1644,)
```

Con estas instrucciones sería suficiente y, por tanto, ya podríamos probar el algoritmo.

Pasamos a probar por primera vez el algoritmo implementado:

```
aprendizaje(1, 50)
```

```
Porcentaje de aciertos:  88.13868613138686 %
```

Como podemos observar para los valores de lambda igual a 1 e iteraciones igual a 50, obtienen un porcentaje de aciertos de 88.14%. Podemos asumir que el algoritmo no es todo lo acertado que podríamos esperar al principio de este trabajo.

Después de hacer varias pruebas cambiando el valor de estos dos parámetros he llegado a la conclusión que los valores que obtienen el mejor porcentaje de aciertos son lambda igual a 1e iteraciones igual a 100.

```
aprendizaje(1, 100)
```

```
Porcentaje de aciertos:  88.9564336372847 %
```

El porcentaje de aciertos no es alto, pero es el mejor resultado obtenido con este algoritmo y conjunto de datos.

3.5. Análisis de los resultados

Como podemos observar la red neuronal, no consigue predecir con un porcentaje de aciertos tan alto como la regresión logística, y esto es debido a la estructura de la red neuronal a la que optamos con estos datos. Por ejemplo, en la práctica 4 los valores de las capas de esta eran 400, 25 y 10 respectivamente, en cambio, la red neuronal que hemos implementado es de 11, 4 y 2. Como en las prácticas de clase solamente hemos utilizado esta estructura, no voy a exponer otros resultados, ya que, el citado anteriormente es el más acertado de esta. Pero cabe señalar que si añado una capa intermedia más o cambio el número de nodos interno de la estructura neuronal el porcentaje de aciertos sube sustancialmente.

4. VECTORES SVM

En esta sección vamos a entrenar un modelo de SVM capaz de predecir y etiquetar los ejemplos de entrenamiento correctamente. Al igual que en los algoritmos anteriores empecemos con la selección de los atributos.

4.1. Elección de los atributos

Guiándome de los resultados obtenidos en la regresión logística, voy a utilizar los mismos tres atributos con los que obtuve el mejor resultado: MSE, Correlación y Disimilitud.

Ya que este algoritmo está relacionado con los algoritmos de regresión y clasificación, espero que estos atributos me lleven a la mejor solución posible. No obstante, en el último apartado de esta sección reevaluaré si es necesario esta u otra selección.

4.2. Implementación de las funciones necesarias

Este algoritmo no tiene ninguna función implementada en su totalidad por mí, ya que lo que necesitamos para este es:

- Funciones para carga y selección de datos (expuestas en la regresión logística)
- Funciones para el entrenamiento del modelo (disponibles en la librería *sklearn.svm*)
- Funciones para la elección de los valores de C y sigma (expuestas en el punto 4.4)

Solo me faltaría exponer las instrucciones necesarias para la correcta ejecución del algoritmo, y estas las voy a mostrar en los apartados siguientes, no sin antes explicar el kernel usado para este modelo.

4.3. Kernel gaussiano

Ya que como pudimos observar en el gráfico de la regresión logística, la clasificación, entre los distintos ejemplos de entrenamiento, no es linealmente separable. Por tanto, he elegido el kernel gaussiano, que se define en la siguiente fórmula:

$$K_{gauss}(x^{(i)}, x^{(j)}) = \exp\left(-\frac{\|x^{(i)} - x^{(j)}\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum_{k=1}^{n}(x_k^{(i)} - x_k^{(j)})^2}{2\sigma^2}\right) = \exp\left(-\gamma\|x^{(i)} - x^{(j)}\|^2\right)$$

Este kernel calcula la distancia entre dos ejemplos de entrenamiento, de tal manera que es capaz de lograr una distinción entre las diferentes etiquetas.

A continuación, voy a mostrar las instrucciones que necesitamos para la ejecución de este algoritmo, hasta el punto en el que estamos, es decir, selección de atributos, entrenamiento del algoritmo y porcentaje obtenido del modelo con el kernel gaussiano.

```
datos = carga_csv('bt_dataset_t3.csv'

datos = seleccionAtributos(datos)
X = datos[:,:-1]
y = datos[:, -1]
print(np.shape(X))
print(np.shape(y))
```

```
(1644, 3)
(1644,)
```

Seleccionamos los tres atributos elegidos, los guardamos en las variables pertinentes, y comprobamos que tenga la forma correcta.

```
#60%
numTrain = 987
#20%
numVal = 328
#20%
numTest = 329
```

Al igual que en la regresión logística elegimos la cantidad de datos que van a formar cada conjunto (entrenamiento, validación y test).

```
xtrain = X[:numTrain,:]
ytrain = y[:numTrain]
print(np.shape(xtrain))
print(np.shape(ytrain))

xval = X[:numVal,:]
yval = y[:numVal]
print(np.shape(xval))
print(np.shape(yval))

xtest = X[:numTest,:]
ytest = y[:numTest]
print(np.shape(xtest))
print(np.shape(ytest))
```

```
(987, 3)
(987,)
(328, 3)
(328,)
(329, 3)
(329,)
```

Dividimos la variable *X* entre los distintos conjuntos, y comprobamos de nuevo la forma.

```
svmAux = svm.SVC(kernel='rbf', C=1, gamma=1/(2*0.1**2))

svmAux.fit(xtrain, ytrain.ravel())
```

```
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=49.99999999999999,
    kernel='rbf', max_iter=-1, probability=False, random_state=None,
    shrinking=True, tol=0.001, verbose=False)
```

Elegimos el kernel que vamos a usar, además de los valores de sigma y C (en este caso C = 1 y sigma = 0.1). Posteriormente, entrenamos el modelo usando los datos de las variables *xtrain* y *ytrain*.

```
svmAux.score(xval, yval)
```

```
0.8658536585365854
```

Por último, comprobamos el resultado de la predicción del modelo entrenado para las variables de validación. Podríamos valorar este resultado con las de los tests, pero como no es el resultado final no nos hace falta. Como vemos el resultado es bastante bajo, y por ello vamos a intentar elegir los valores más adecuados de C y sigma, y reintentarlo de nuevo.

4.4. Elección de los parámetros c y sigma

Para seleccionar el valor de estos parámetros he implementado la siguiente función:

```
def calcularPorcentaje (C, Sigma, x, y, xval, yval):
    adecuado = 0

    for c in C :
        for sigma in Sigma :
            svmAux = svm.SVC(kernel='rbf', C=c, gamma=1/(2*sigma**2))
            svmAux.fit(x, y)
            porc = svmAux.score(xval, yval)

            if(adecuado < porc) :
                adecuado = porc
                cSigma = (c, sigma)

    return cSigma
```

Esta función recibe un array de posibles valores, tanto de sigma como de C, los datos de entrenamiento y de validación. Realiza dos bucles, uno de ellos anidado sobre el otro, con los que prueba, siguiendo los pasos mostrados en el apartado anterior, cada valor de sigma y C comprobándolos con los datos de validación y eligiendo los que obtienen un mayor porcentaje de aciertos en las predicciones.

A continuación, muestro las tres instrucciones necesarias para obtener los valores que estamos buscando.

```
aux = np.array([0.01,0.03,0.1,0.3,1,3,10,30])
cSigma = calcularPorcentaje(aux, aux, xtrain, ytrain.ravel(), xval, yval.ravel
cSigma
```

```
(30.0, 0.01)
```

Como vemos los valores de C y sigma correctos son 30 y 0.01 respectivamente. Los valores elegidos en el array son los mismo que estudiamos en la práctica 6, una vez que comprobemos el resultado en el siguiente apartado del modelo, reevaluaré si hace falta probar con valores distintos.

4.5. Prueba del algoritmo con los test

Una vez que hemos calculado los valores adecuados de C y sigma, procedemos a comprobar el modelo entrenado con los ejemplos dedicados a los test.

```
svmAux = svm.SVC(kernel='rbf', C=cSigma[0], gamma=1/(2*cSigma[1]**2))

svmAux.fit(xtrain, ytrain.ravel())

svmAux.score(xval, yval)
```

```
0.9908536585365854
```

```
svmAux.score(xtest, ytest)
```

```
0.9908814589665653
```

Como podemos observar el resultado del modelo entrenado ha obtenido buenos resultados tanto con los datos de validación (era de esperar tras la ejecución del apartado anterior) como con los de test.

4.6. Análisis de los resultados

Podemos observar que el error de este modelo es el más pequeño que hemos obtenido en todo el estudio, por tanto, creo innecesario tanto el reevaluar datos diferentes para Sigma y C como elegir otro conjunto de datos con el que comprobar este modelo.

Doy por cerrado el estudio de este algoritmo y procedo a exponer las conclusiones que he sacado de este trabajo.

5. CONCLUSIÓN

La primera conclusión y en mi opinión la más obvia, es la confirmación de lo que auguraba en la introducción, y es que, para este conjunto de datos, el mejor algoritmo para realizar predicciones es el de vectores SVM.

Una vez visto todos los resultados obtenidos podríamos descartar la red neuronal (con la estructura neuronal expuesta), ya que su predicción es >90% y tanto el de regresión logística como el de vectores SVM es <95%.

Por último, para cerrar este estudio, lo que más me ha sorprendido es que la parte más importante y en la que he observado que más cambios ocasionaba en los resultados de cada algoritmo, es la selección de los atributos, ya que he podido comprobar como un atributo más o menos ocasionaba una diferencia de más de 15% en el porcentaje de aciertos.