Universidad Autónoma de Querétaro

Facultad de Ingeniería
División de Investigación y Posgrado

Reporte 4

Regresión Logística a Dataset Digits

Maestría en Ciencias en Inteligencia Artificial
Optativa de especialidad II - Deep Learning

Aldo Cervantes Marquez
Expediente: 262775
Profesor: Dr. Sebastián Salazar Colores

Santiago de Querétaro, Querétaro, México
Semestre 2022-2
29 de Agosto de 2022

Índice

1. Introducción

La presente práctica consiste en realizar una regresión logística, aplicar gradiente descendente y dividir los datos de la base de datos en secciones, con el fin de poder observar los comportamientos del aprendizaje y aplicar los conceptos de prácticas pasadas. La base de datos consiste en una serie de imágenes en la libreria *klearn.datasets* y se llama *digits*.

1.1. Regresión Logística

Consiste en una función sigmoidal que tiene la siguiente forma (Figura 1):

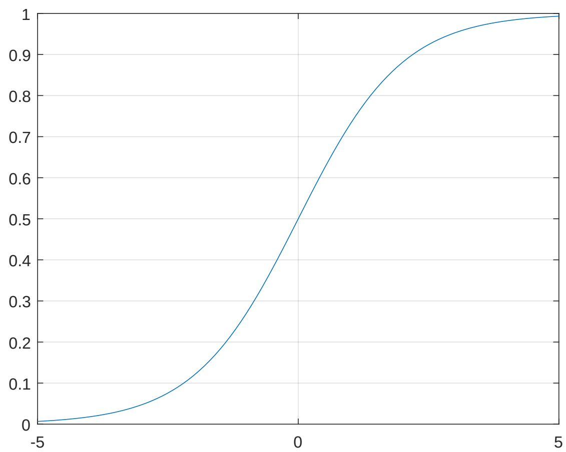


Figura 1: Función sigmoide.

Se representa matemáticamente como:

$$h_\theta(z) = \frac{1}{1+e^{-z}} \tag{1}$$

Donde:

$$z = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n \tag{2}$$

De igual modo, se deberá realizar un ajuste de las ganancias θ_n para poder realizar un ajuste correcto, y una predicción de la clasificación adecuada.

Para poder realizar la clasificación y aplicando una propiedad que tiene la función sigmoidal, la cual consiste en que todos sus valores se acercan entre 0 y 1, se va a truncar los valores de la siguiente manera:

$$h_\theta(z) = \begin{cases} 0 \quad si \quad h_\theta(z) < 0.5 \\ \\ 1 \quad si \quad h_\theta(z) \geq 0.5 \end{cases} \tag{3}$$

1.2. Forma Matricial de z

Partiendo de la ecuación (3) podemos acomodar las variables θ_0 y θ_1 de la siguiente manera:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \tag{4}$$

Del mismo modo podemos agrupar las características de x en cualquier dimensión a partir de la siguiente matriz:

$$X = \begin{bmatrix} x_1^0 & x_1^1 & . & . & x_1^n \\ x_2^0 & x_2^1 & . & . & x_2^n \\ . \\ . \\ . \\ x_m^0 & x_m^1 & . & . & x_m^n \end{bmatrix} \tag{5}$$

Donde $x^0_{1,2,3,..,n}$ es igual a 1 puesto que representa al termino independiente de la ecuación (θ_0). Por lo que generalizando y por producto de matrices, es posible simplificar la operación de la siguiente manera la hipótesis h_θ:

$$h_\theta = \begin{bmatrix} x_1^0 & x_1^1 & . & . & x_1^n \\ x_2^0 & x_2^1 & . & . & x_2^n \\ . \\ . \\ . \\ x_m^0 & x_m^1 & . & . & x_m^n \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ . \\ . \\ . \\ \theta_n \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ . \\ . \\ . \\ h_m \end{bmatrix} \tag{6}$$

1.3. Cálculo del error

Existen criterios para observar que tan bueno es el ajuste hecho, y se basan en los errores residuales de los datos disponibles evaluados en los puntos que se tienen. Se tiene el error medio cuadrático (MSE), el cual se muestra a continuación.

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2 \tag{7}$$

También se tiene el error absoluto medio (MAE).

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} |h_\theta(x_i) - y_i| \tag{8}$$

Donde h_0 es la predicción realizada en el valor x_i

La idea principal es minimizar este error a 0 y por lo tanto tener el mejor ajuste posible.

1.4. Gradiente Descendente

El uso de gradiente descendente es método para obtener valores mínimos de funciones mediante el uso de criterios de derivadas (criterio de primera derivada) donde se sabe que los ceros (o raíces) de una función (véase Figura 2).

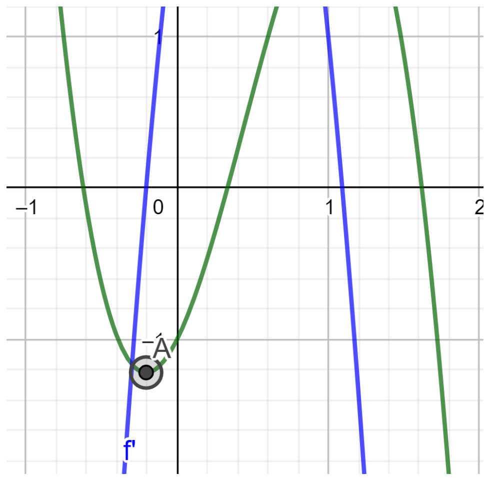


Figura 2: Criterio de derivada.

Como se observa, la función verde $f(x)$ en el punto A se tiene un mínimo y en ese mismo punto se tiene en la función derivada azul $f'(x)$ tiene un cruce con el eje x.

A continuación, se muestra un pseudocódigo del algoritmo desarrollado.

Algoritmo 1 Proceso de gradiente descendiente.

Inicio
Valor arbitrario de θ_0 y θ_1
Repetir (**hasta** que se cumpla **condicion** de paro)
Buscar dirección $\boldsymbol{v}$ de decrecimiento en el campo $\boldsymbol{x}$
Variación valores de la dirección
$\boldsymbol{x} \rightarrow \boldsymbol{x} + \boldsymbol{\varepsilon v}$

Gráficamente se puede observar como la tasa de cambio de la derivada y su movimiento a través de la función desplazándose cierta cantidad de unidades hacia el mínimo (véase Figura 3).

Matemáticamente y computacionalmente se puede definir la formula para un numero de épocas (iteraciones) determinado.

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \tag{9}$$

donde α es la taza de aprendizaje.

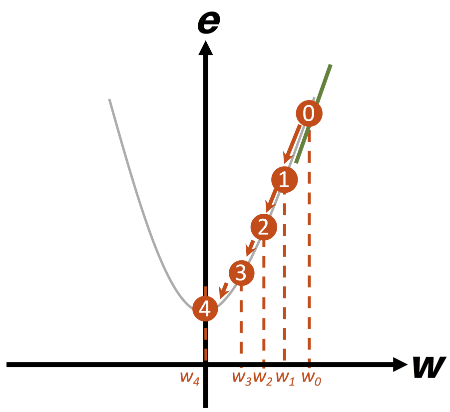


Figura 3: Criterio de derivada.

1.5. Gradiente Descendiente con matrices.

En este caso se usará el gradiente descendiente con el criterio de minimizar la función de costo $J(\theta)$ que será el MSE mencionado anteriormente.

Sabiendo la derivada de MSE se obtiene la ecuación (10)

$$\begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ . \\ . \\ . \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} = \frac{1}{m}(h_\theta - Y)X^T = \frac{1}{m}(X.\theta - Y)X^T \tag{10}$$

La cual describe la derivada parcial con respecto a cada θ_n. Debido a las operaciones matriciales, es más cómodo calcular para n dimensiones de θ.

La formula completa implica el uso de la actualización de θ, por lo que al agregarle el factor α se obtiene la ecuación (11).

$$\begin{bmatrix} \theta_0 \\ \theta_1 \\ . \\ . \\ . \\ \theta_n \end{bmatrix} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ . \\ . \\ . \\ \theta_n \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ . \\ . \\ . \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} \tag{11}$$

1.6. Base de datos

La base de datos consiste en dos matrices, la primera de entrada (matriz x), la cual tiene una dimensión de 1797x64 y otra de salida (matriz y) que tiene una dimensión de 64x1. En la matriz x se tienen en las filas los casos de cada arreglo de la imagen, y en cada columna es el valor de cada píxel en formato de 2^4 dígitos (16). Por otro lado, se tiene una sola columna con 64 filas y cada elemento es el elemento de salida o correspondencia a cada arreglo de la matriz x en la posición n (véase Figura 4).

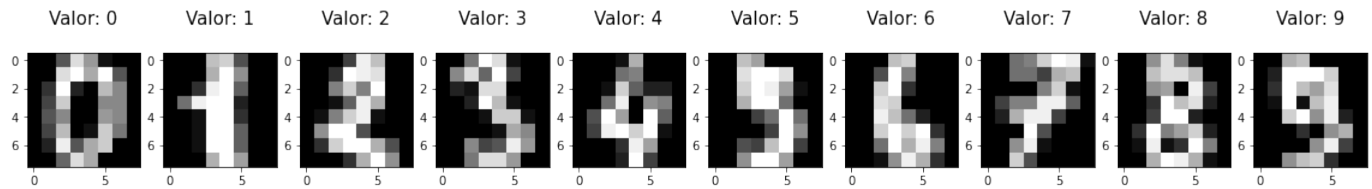


Figura 4: Imágenes de la base de datos.

1.7. División de los Datos

Con el fin de poder realizar un análisis más completo y evitar sesgos en los resultados, es conveniente realizar divisiones aleatorias de los datos para poder saber si verdaderamente el modelo propuesto esta funcionando correctamente. Por lo que a continuación se ocuparan 2 métodos para poder segmentar la información [1]. Definimos la precisión como:

$$p = \frac{aciertos}{totales} \times 100\,\% \tag{12}$$

1.7.1. Regla 60 20 20

Esta regla consiste en dividir los datos en un 60 % para entrenar y aplicar gradiente descendiente. Un 20 % para verificar los datos y de este modo tener un valor de precisión inicial, finalmente el otro 20 % se utiliza para las pruebas finales y dar una precisión final (véase Figura 7).

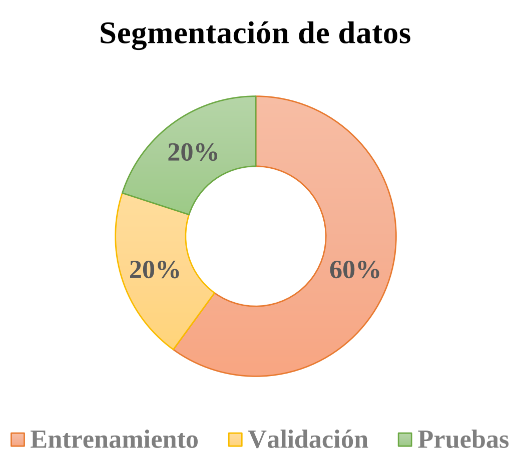


Figura 5: División de datos por porcentajes.

1.7.2. K-Folds

Este método consiste en crear k grupos aleatorios de igual tamaño para poder realizar una combinación de las precisiones y obtener un promedio. De este modo es posible observar si existen datos con los que aprenda mejor el modelo o no (véase Tabla 1).

Tabla 1: Grupos de K-folds

	Grupos				
Combinaciones	1_1	1_2	1_3	$\dots$	1_k
	2_1	2_2	2_3	$\dots$	2_k
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	C_1	C_2	C_3	$\dots$	C_k
	Grupo de prueba				
	Grupos de entrenamiento				

finalmente se aplica la media de los valorees obtenidos de la precisión:

$$p(comb) = \frac{1}{c}\sum_{i=1}^{c} p(i) \tag{13}$$

2. Justificación

El uso de funciones sigmoidales permiten que se pueda realizar una clasificación muy marcada entre valores y permite categorizar entre dos grupos, logrando que se pueda identificar de una manera más directa la pertenencia de cada grupo.

3. Resultados

Para poder visualizar los resultados, se explicarán los pasos realizados con imagenes de los resultados o el material de apoyo requerido.

1. Se obtuvieron los datos normalizados de la base de datos.
2. Se dividieron los datos en entradas y salidas.
3. Se buscaron 2 números aleatorios diferentes para poder clasificarlos, debido a la naturaleza de la función $h_\theta(z)$ (en este caso fue el número 5 y 1).
4. se separaron los valores que contenían a dichos números.

5. Se normalizaron los valores 5 y 1 de la matriz y de tal modo que $5 \to 1$ y $1 \to 0$.

6. Se observó que los datos fueron 364x64 y se agregó la columna de 1's para obtener la forma de la ecuación (5) incrementando a 364x65.

7. Se propusieron los valores de θ_n con $n = 65$ de manera aleatoria.

8. Se realizó la obtención de $X.\theta = h_\theta(z) = \sigma(z)$ para obtener la ecuación con la función sigmoidal de manera matricial como en las ecuaciones (1) y (6).

9. Se realizó una prueba de gradiente descendiente, sin división de datos, obteniendo lo siguiente:

Tabla 2: Resultados del entrenamiento con el 100 % de los datos.

Numero de datos	Numero de θ_n	epocas	alpha	MSE final
365	65	80	0.1	0.0001

Obteniendo el siguiente comportamiento:

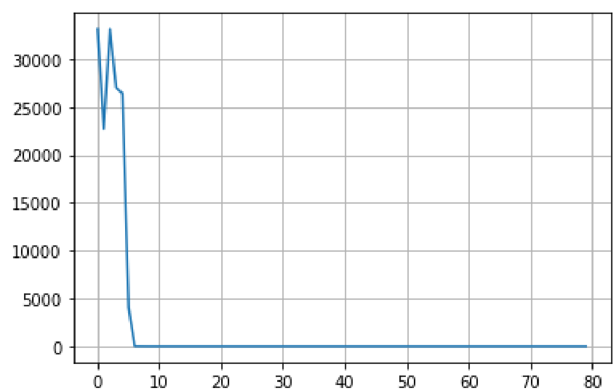


Figura 6: Comportamiento de MSE con 100 % de los datos.

10. Posteriormente se probó la división de datos con el método de 60 20 20, con la función en el código llamada como *def div_data()*. Mediante el uso de ciclos for se obtuvo una división de los porcentajes que se solicitan. Obteniendo los siguientes resultados partiendo de $\theta's$ aleatorias.

Tabla 3: Resultados del entrenamiento con el 60 20 20 % de los datos.

Numero de datos	Numero de θ_n	épocas	alpha	Precisión Verificación	Precisión prueba	MSE final
364	65	50	0.01	100%	98.63%	$1.8121\text{x}10^{-05}$

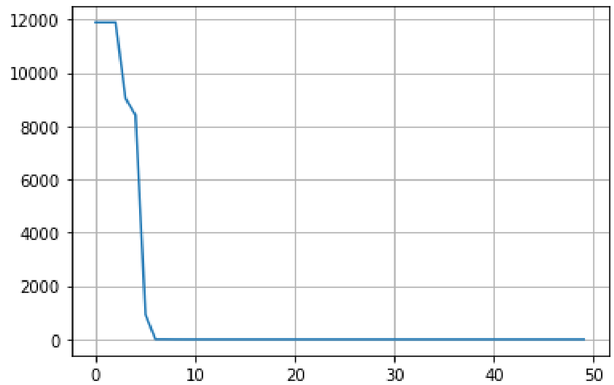


Figura 7: Comportamiento de MSE entrenamiento con 60 20 20 de los datos.

11. Posteriormente se aplicó el método de K-folds para poder separar los valores de los grupos y se aplica la verificación para cada combinación del grupo. Obteniendo los siguientes resultados.

Tabla 4: Resultados del entrenamiento con el 60 20 20 % de los datos.

	Numero de datos	# de k-Folds	Numero de θ_n	épocas	alpha	Precisión Verificación	MSE final
	364	5	65	50	0.01	100%	0.000296
Prueba	# Datos entrenamiento / verificación		Resultados individuales				
1	292	72	65	50	0.01	100%	3.148 e − 5
2	292	72	65	50	0.01	100%	*0.000872*
3	292	72	65	50	0.01	100%	*4.493e-5*
4	292	72	65	50	0.01	100%	*3.8508e-5*
5	292	72	65	50	0.01	100%	*0.00049*

4. Conclusiones

El uso de regresión logística con la función σ es muy útil para la clasificación binaria entre dos grupos (en este caso el patrón de imágenes de 2 números), en donde se observó un buen comportamiento del modelo, siendo muy efectivo y por lo tanto por las dimensiones del problema, podría servir para muchos otros más. Se trató de simplificar en funciones para reciclar código y poder generalizar para futuras prácticas.

Referencias

[1] "Train Test Validation Split: How To & Best Practices [2022]."

Práctica 4 clasificación

August 28, 2022

[364]:
```
from sklearn.datasets import load_digits
digits = load_digits()
```

[365]:
```
print(digits.data.shape)
print(digits.target.shape)
```

```
(1797, 64)
(1797,)
```

[366]:
```
X = digits.data
Y = digits.target

print(X, len(X))
print(Y, len(Y))
```

```
[[ 0.  0.  5. ...  0.  0.  0.]
 [ 0.  0.  0. ... 10.  0.  0.]
 [ 0.  0.  0. ... 16.  9.  0.]
 ...
 [ 0.  0.  1. ...  6.  0.  0.]
 [ 0.  0.  2. ... 12.  0.  0.]
 [ 0.  0. 10. ... 12.  1.  0.]] 1797
[0 1 2 ... 8 9 8] 1797
```

[367]:
```
import numpy as np
import matplotlib.pyplot as plt


plt.figure(figsize=(20,4))

inicial = 0

for index, (imagen, etiqueta) in enumerate(zip(X[inicial:inicial+10], Y[inicial:
 ↪inicial+10])):
 plt.subplot(1, 10, index + 1)
 plt.imshow(np.reshape(imagen, (8,8)), cmap=plt.cm.gray)
 plt.title('Valor: %i\n' % etiqueta, fontsize = 15)
```

Programar un clasificador utilizando regresión logística

[368]:
```
import random
valores=np.random.randint(0,10,size=(1,2))
#valores=np.array([1, 4])
valores=sum(valores)
#print(valores)
while valores[0]==valores[1]:
    valores=np.random.randint(0,9,size=(1,2))
    valores=sum(valores)

print(valores)
```

```
[5 1]
```

[369]:
```
valy=np.where((Y==valores[0]) | (Y==valores[1]))
valx=X[valy]
#print(valx)
[renx,colx]=valx.shape
valy_f=np.asarray(Y[valy])

valy_f=valy_f.T
print(valy_f.shape)
print(valx.shape)
print(valy_f)
```

```
(364,)
(364, 64)
[1 5 1 5 1 5 5 5 5 1 5 1 1 1 5 5 1 1 1 1 1 5 1 5 5 5 1 5 1 5 1 5 5 5 5 1 5
 1 1 1 5 5 1 1 1 1 1 5 1 5 5 5 1 5 1 5 1 5 5 5 5 1 5 1 1 1 5 5 1 1 1 1 1 5
 1 5 5 5 1 5 1 5 1 5 5 5 5 1 5 1 1 1 5 5 1 1 1 1 1 5 1 5 5 5 1 5 1 5 1 5 5
 5 5 1 5 1 1 1 5 5 1 1 1 1 1 5 1 5 5 5 1 5 1 5 1 5 5 5 5 1 5 1 1 1 5 5 1 1
 1 1 1 5 1 5 5 5 1 5 1 5 1 5 5 5 5 1 5 1 1 1 5 5 1 1 1 1 1 5 1 5 5 5 1 5 1
 5 1 5 5 5 5 1 5 1 1 1 5 5 1 1 1 1 1 5 1 5 5 5 1 5 1 5 1 5 5 5 5 1 5 1 1 1
 5 5 1 1 1 1 1 5 1 5 5 5 1 5 1 5 1 5 5 5 5 1 5 1 1 1 5 5 1 1 1 1 1 5 1 5 5
 5 1 5 1 5 1 5 5 5 5 1 5 1 1 1 5 5 1 1 1 1 1 5 1 5 5 5 1 5 1 5 1 5 5 5 5 1
 5 1 1 1 5 5 1 1 1 1 1 5 1 5 5 5 1 5 1 5 1 5 5 5 5 1 5 1 1 1 5 5 1 1 1 1 1
 5 1 5 5 5 1 5 1 5 1 5 5 5 5 1 5 1 1 1 5 5 1 1 1 1 1 5 1 5 5 5]
```

Agregar columna de 1s

[370]:
```
valx=np.insert(valx,0,np.ones([len(valx)],dtype=int),axis=1)

[renx,colx]=valx.shape
print(renx,colx)
```

```
364 65
```

Cambiar etiquetas de Y por 0s y 1s

[371]:
```
print(valy_f.shape)
for q in range(renx):
  if valy_f[q]==valores[0]:
    valy_f[q]=0
  else:
    valy_f[q]=1

valy_f=np.reshape(valy_f,(renx,1))
print(valy_f.shape)
```

```
(364,)
(364, 1)
```

Proponer thetas aleatorias

[372]:
```
thetas=np.random.rand(colx,1)
print(thetas.shape)
```

```
(65, 1)
```

Calcular la funcion $X.\theta$

[373]:
```
xt=np.dot(valx,thetas)
xt.shape
```

[373]: (364, 1)

proponer sigma

[374]:
```
def sigma(xport): #Hipótesis
  s=1/(1+np.exp(-(xport)))
  return s

print(sigma(xt).shape)
```

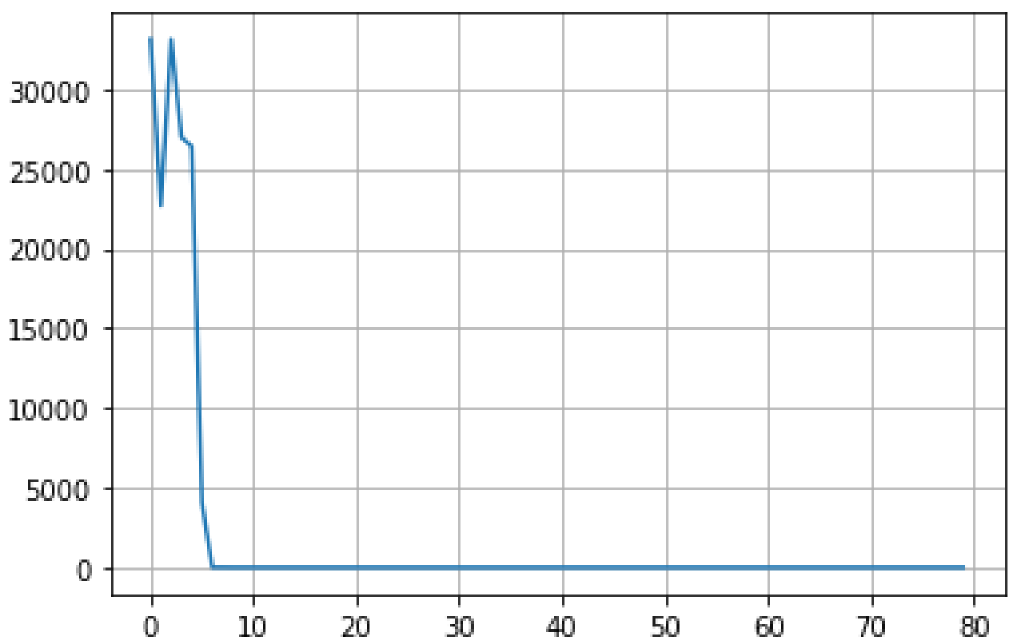
```
(364, 1)
```

Aplicar gradiente con matrices caso general.

[375]:
```
import matplotlib.pyplot as plt
epocas=80
alpha=0.1
m=renx
```

```
print(xt.shape,valy_f.shape)
print(valx.shape)
msemat=[]
epocmat=[]
for epoca in range(epocas):    ##
  thetas=thetas-alpha*(1/m)*((sigma(xt)-valy_f).T.dot(valx)).T #Importante para␣
 ↪que funcionara
  mse=sum(sigma(xt)-valy_f)**2
  msemat.append(mse)
  epocmat.append(epoca)
  xt=np.dot(valx,thetas)
  #print(mse)
plt.plot(epocmat,msemat)
plt.grid()
print(msemat[-1])
```

```
(364, 1) (364, 1)
(364, 65)
[0.00010981]
```



1 Parte 2: entrenamiento con división de valores 60 20 20

[376]:
```
#aplicando regla de 60 20 20
#Cambiando variables
X_r1=valx# X
thetas_r1=thetas# Theta
y_r1=valy_f# y
[renx_r1,colx_r1]=X_r1.shape
m_r1=renx_r1# m

print(X_r1.shape)
print(y_r1.shape)
import random

#division de valores mediante quitar y copiar valores
def div_data(x,y,porcentajes):
  x_total=x
  [rowx,colx]=x_total.shape
  y_total=y
  ll=rowx
  x_train=np.empty((0,colx),dtype=float)
  y_train=np.empty((0,1),dtype=float)
  x_ver=np.empty((0,colx),dtype=float)
  y_ver=np.empty((0,1),dtype=float)
  x_test=np.empty((0,colx),dtype=float)
  y_test=np.empty((0,1),dtype=float)
  ##Datos de entrenamiento
  for ii in range(round(ll*porcentajes[0]/100)):
    p=random.randint(0,len(x_total)-1)
    ## Para x
    val_el=x_total[p,:]
    x_total=np.delete(x_total,p,0)
    x_train=np.vstack([x_train,val_el])
    ##para y
    val_el=y_total[p,:]
    y_total=np.delete(y_total,p,0)
    y_train=np.vstack([y_train,val_el])
  ## Datos de verificación
  for ii in range(round(ll*porcentajes[1]/100)):
    p=random.randint(0,len(x_total)-1)
    ## Para x
    val_el=x_total[p,:]
    x_total=np.delete(x_total,p,0)
    x_ver=np.vstack([x_ver,val_el])
    ##para y
    val_el=y_total[p,:]
    y_total=np.delete(y_total,p,0)
```

```
    y_ver=np.vstack([y_ver,val_el])

  ## La parte de test, es el resultado de x y y total
  x_test=x_total
  y_test=y_total

  return [x_train,y_train,x_ver,y_ver,x_test,y_test]

[x_train,y_train,x_ver,y_ver,x_test,y_test]=div_data(X_r1,y_r1,[60, 20, 20])
```

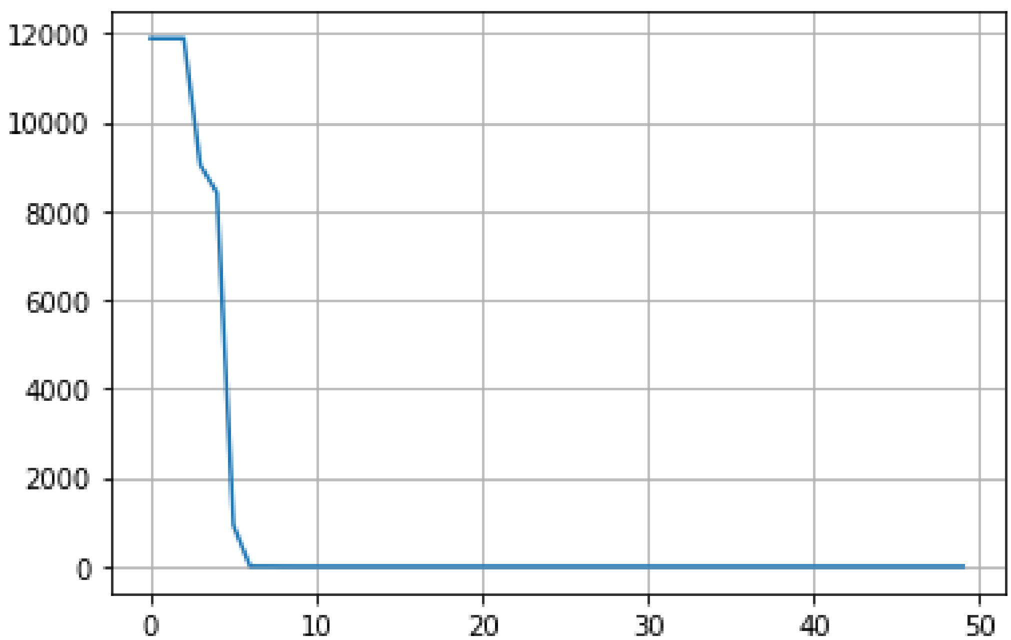
```
(364, 65)
(364, 1)
```

aplicación de Gradiente descendiente para 60 20 20

[377]:
```
alpha_r1=0.01# Alpha
epocas=50
t_1=thetas=np.random.rand(colx_r1,1)
def g_epocas(x,y,thetas,epocas,apha):  ##Entrenamiento
  msemat=[]
  epocmat=[]
  [rowx,colx]=x.shape
  mse=0
  xt=np.dot(x,thetas)
  m=len(x)
  for epoca in range(epocas):
    thetas=thetas-alpha*(1/m)*((sigma(xt)-y).T.dot(x)).T
    mse=sum(sigma(xt)-y)**2
    msemat.append(mse)
    epocmat.append(epoca)
    xt=np.dot(x,thetas)
  plt.plot(epocmat,msemat)
  plt.grid()
  print('Resultados del entrenamiento MSE: ',float(msemat[-1]))
  return thetas

def prueba(x,y,thetas):
  acc=0
  aciertos=0
  posibles=len(x)
  xt=np.dot(x,thetas)
  for ii in range(len(x)):

    p_r=sigma(xt[ii,:])
    #print(p_r)
    if p_r>0.5:
      p_r=1
    else:
```

```
      p_r=0

    if p_r==y[ii,0]:
      aciertos+=1

  acc=(aciertos/posibles)*100
  return acc






thetas_fin=g_epocas(x_train,y_train,t_1,epocas,alpha) #Thetas despues de␣
 ↪entrenamiento
print('Los resultados de la verificación son:␣
 ↪',prueba(x_ver,y_ver,thetas_fin),'% de precisión')
print('Los resultados de test son: ',prueba(x_test,y_test,thetas_fin),'% de␣
 ↪precisión')
```

```
Resultados del entrenamiento MSE:  1.8121885665324867e-05
Los resultados de la verificación son:  100.0 % de precisión
Los resultados de test son:  98.63013698630137 % de precisión
```



2 Método K-Folds con K=4

[381]:
```
#aplicando regla k-folds
#Cambiando variables
X_r2=valx# X
y_r2=valy_f# y
k=5  ### Cantidad de divisiones
def k_folds(x,y,k):  #Función que únicamente divide los grupos
  x_total=x
  [rowx,colx]=x_total.shape
  y_total=y
  ll=rowx
  x_group=np.empty((0,colx),dtype=float)
  y_group=np.empty((0,1),dtype=float)
  resx=[]
  resy=[]
  for div in range(k):
    x_group=np.empty((0,colx),dtype=float)
    y_group=np.empty((0,1),dtype=float)
    for ii in range(round(np.trunc(len(x)/k))):
      p=np.random.randint(0,len(x_total))
      val_el=x_total[p,:]   #Para x
      x_total=np.delete(x_total,p,0)
      x_group=np.vstack([x_group,val_el])
      ##Para y
      val_el=y_total[p,:]
      y_total=np.delete(y_total,p,0)
      y_group=np.vstack([y_group,val_el])
    resx.append(x_group)
    resy.append(y_group)
  return [resx,resy]

[g_x,g_y]=k_folds(X_r2,y_r2,k)
print(len(g_x))
```

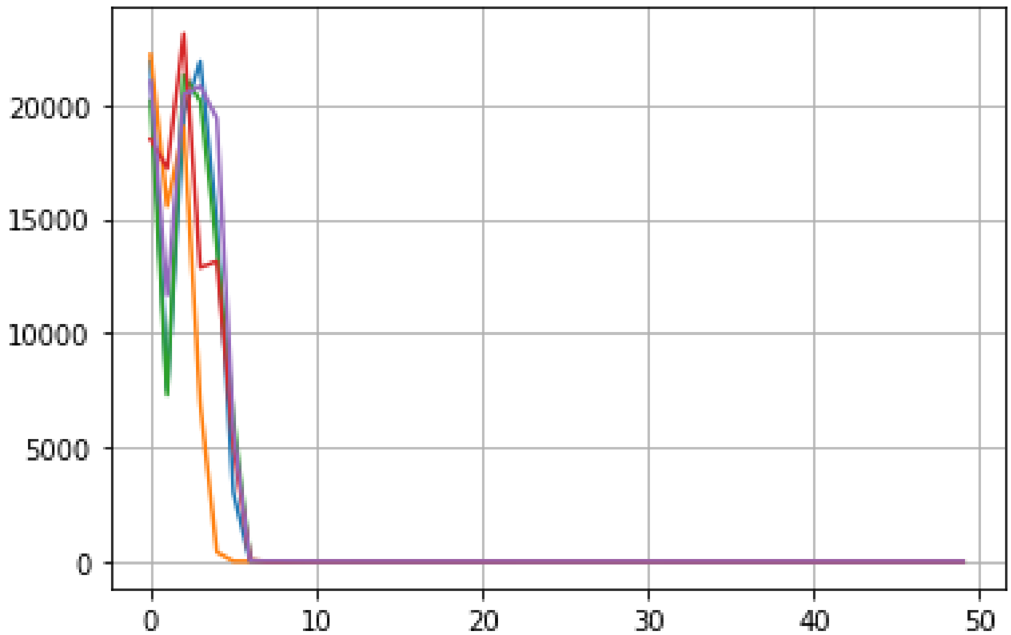
```
5
```

3 Muestra de resultados de K-folds

[382]:
```
alpha_r1=0.001# Alpha
epocas=50
t_1=thetas=np.random.rand(colx_r1,1)
[renx_r2,colx_r2]=X_r1.shape
m_r2=renx_r2# m

########## Prueba de valores de K ###
for rr in range(k):
```

```
  gr_tempx=g_x
  gr_tempy=g_y
  gr_verx=gr_tempx[rr]
  gr_very=gr_tempy[rr]
  gr_trainy=np.empty((0,1),dtype=float)
  gr_trainx=np.empty((0,colx),dtype=float)
  for eliminate in range(k):
    if eliminate==rr:
      print('segmento omitido',eliminate)
    else:
      gr_trainx=np.vstack([gr_trainx,gr_tempx[eliminate]])
      gr_trainy=np.vstack([gr_trainy,gr_tempy[eliminate]])

  t_1=thetas=np.random.rand(colx_r1,1)
  t_n=g_epocas(gr_trainx,gr_trainy,t_1,epocas,alpha)
  prueba(gr_verx,gr_very,t_n)
  print('Los resultados de la verificación son:␣
↪',prueba(x_ver,y_ver,thetas_fin),'% de precisión')
```

```
segmento omitido 0
Resultados del entrenamiento MSE:  3.148296868423244e-05
Los resultados de la verificación son:  100.0 % de precisión
segmento omitido 1
Resultados del entrenamiento MSE:  0.0008728879248555085
Los resultados de la verificación son:  100.0 % de precisión
segmento omitido 2
Resultados del entrenamiento MSE:  4.493979357609134e-05
Los resultados de la verificación son:  100.0 % de precisión
segmento omitido 3
Resultados del entrenamiento MSE:  3.8508174172384236e-05
Los resultados de la verificación son:  100.0 % de precisión
segmento omitido 4
Resultados del entrenamiento MSE:  0.0004945740365993505
Los resultados de la verificación son:  100.0 % de precisión
```



[380]:
```
los=np.array(g_x[1])
print(los.shape)
```

```
(91, 65)
```