Universidad de Huelva

Universidad de Huelva

Grado en Ingeniería Informática

Regresión Lineal

Memoria de Prácticas

Autor: Alberto Fernández Merchán
Asignatura: Aprendizaje Automático

Índice

1. Introducción

En esta práctica implementaremos varios métodos de **regresión lineal** utilizando el conjunto de datos (*regresion1.csv*) que hay subido en la página de la asignatura.

Los métodos que vamos a usar para calcular los coeficientes θ de la recta de regresión serán:

- **Aplicación Externa**: Utilizando la librería *sklearn* de Python calcularemos los coeficientes de la recta de regresión lineal para tener una primera aproximación.
- **Ecuaciones Normales**: Utilizaremos las ecuaciones normales para calcular los coeficientes de una manera simple, sencilla e inmediata.
- **Descenso Por Gradiente**: Utilizaremos el algoritmo de descenso por gradiente para calcular una aproximación de los coeficientes de la recta de regresión lineal. Además, calcularemos los valores que toma $J(\theta)$ para comprobar que el error decrece.
- **Descenso Por Gradiente Estocástico**: Utilizaremos una variante estocástica del algoritmo de descenso por gradiente para que el proceso de cálculo de los coeficientes sea más rápido y evitemos caer en mínimos locales.

Hay que recordar que para calcular la recta de regresión debemos añadir al dataset **un atributo más** que esté establecido a $x_0 = 1$.

2. Representación Gráfica

Primero debemos representar los datos para hacernos una idea de su distribución.
Realizaré un gráfico de dos dimensiones utilizando el lenguaje de programación *Python*:

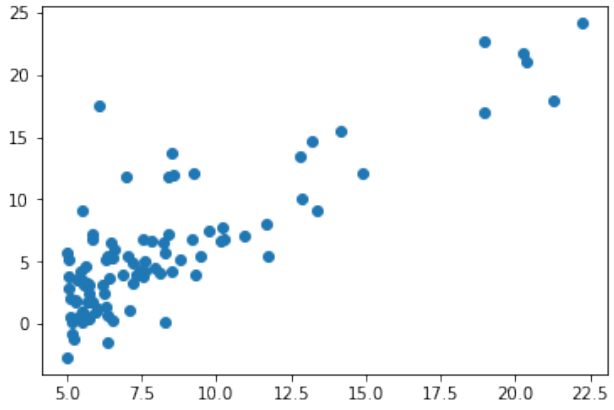


Figura 1: Distribución de los puntos

En la figura 1 podemos ver una clara tendencia ascendente del conjunto de los datos. Podemos intuir una dependencia entre X e Y.

3. Aplicación Externa: sklearn

Mediante el módulo de **modelo lineal** de la biblioteca **sklearn** podemos sacar los coeficientes de la recta de regresión lineal que podría modelar los datos que tenemos en el fichero proporcionado.

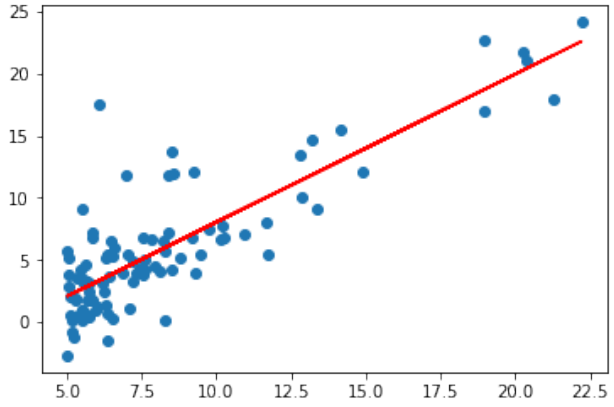


Figura 2: Recta de regresión generada mediante la librería sklearn de Python

Los coeficientes que proporciona dicha biblioteca son los siguientes:

$$y = \theta_1 \cdot x_1 + \theta_0 \cdot x_0 \rightarrow y = 1{,}1930336441895932 \cdot x_1 - 3{,}8957808783118484 \cdot x_0 \qquad (x_0 = 1)$$

4. Ecuaciones Normales

Aplicaremos la definición analítica de $\vec{\theta}$ para calcular sus coeficientes. Este método no se puede usar en todas las situaciones, sin embargo, en los datos que nos proporciona el enunciado sí se puede aplicar.

La ecuación que debemos utilizar es la siguiente:

$$\vec{\theta} = (X^T X)^{-1} X^T \vec{y}$$

Debemos tener en cuenta que, si la inversa de la matriz $X^T X$ no existe, entonces debemos utilizar otro método (descenso por gradiente). En este caso, como he mencionado antes, sí existe dicha inversa y, por tanto, podemos aplicar la definición analítica.

Utilizaremos el módulo de álgebra lineal de la biblioteca *numpy* del lenguaje *Python*. El código es el siguiente:

```
def calcularCoeficientesEcNormal(X,Y):
  coeficientes = -1
  newX = np.hstack((np.ones((X.shape[0], 1)),X))
  # Calculo el determinante para saber si se puede hacer la inversa.
  determinante = np.linalg.det(newX.T.dot(newX))

  if (determinante != 0.0):
    # Si el determinante no es nulo, entonces podemos calcular los coeficientes.
    coeficientes = np.linalg.inv(newX.T.dot(newX)).dot(newX.T).dot(Y)

  return coeficientes
```

El resultado que nos da este método es el siguiente:

$$y = \theta_1 \cdot x_1 + \theta_0 \cdot x_0 \rightarrow y = 1{,}19303364 \cdot x_1 - 3{,}89578088 \cdot x_0 \qquad (x_0 = 1)$$

5. Descenso Por Gradiente

En los casos donde no se pueda utilizar el método de las ecuaciones normales porque no se pueda realizar la inversa de la matriz, podemos utilizar el algoritmo de **descenso por gradiente**. Este algoritmo aproxima, sucesivamente, el resultado de los coeficientes utilizando el concepto de **gradiente**.

En este apartado mostraremos gráficamente el error total de cada iteración y comprobaremos que se genera una gráfica monótona decreciente. La fórmula del error es la siguiente:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^j) - y^j)^2$$

En nuestro caso, el algoritmo realizará 1000 iteraciones y lo probaremos con tres coeficientes de aprendizaje (α) diferentes. Dividiremos el coeficiente de aprendizaje entre el número de instancias que vamos a utilizar ($m = 97$) para trabajar sobre el error relativo:

- $\alpha = \mathbf{0{,}01/97}$:
  La ecuación obtenida con este parámetro es la siguiente:

  $$y = \theta_1 \cdot x_1 + \theta_0 \cdot x_0 \rightarrow y = 1{,}19303345 \cdot x_1 - 3{,}89577897 \cdot x_0 \qquad (x_0 = 1)$$

  La gráfica del error ($J(\theta)$) evoluciona de la siguiente forma:

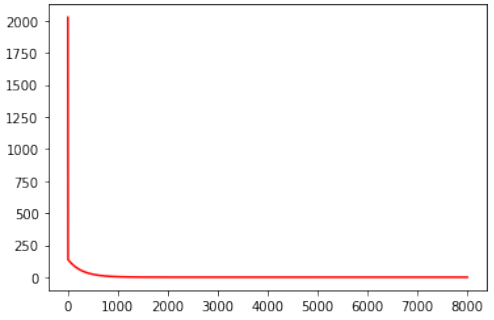


Figura 3: Error con $\alpha = 0{,}00010309278350515464$

- $\alpha = \mathbf{0{,}001/97}$:
  La ecuación obtenida con este parámetro es la siguiente:

  $$y = \theta_1 \cdot x_1 + \theta_0 \cdot x_0 \rightarrow y = 1{,}09874303 \cdot x_1 - 2{,}95703245 \cdot x_0 \qquad (x_0 = 1)$$

  La gráfica del error ($J(\theta)$) evoluciona de la siguiente forma:

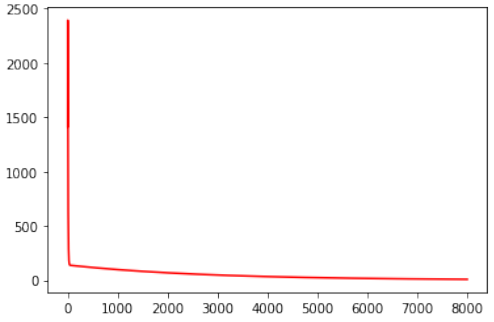


Figura 4: Error con $\alpha = 1{,}0309278350515464e - 05$

- $\alpha = \mathbf{0{,}0001/97}$:
  La ecuación obtenida con este parámetro es la siguiente:

$$y = \theta_1 \cdot x_1 + \theta_0 \cdot x_0 \rightarrow y = 0{,}84729442 \cdot x_1 - 0{,}4541873 \cdot x_0 \qquad (x_0 = 1)$$

  La gráfica del error ($J(\theta)$) evoluciona de la siguiente forma:

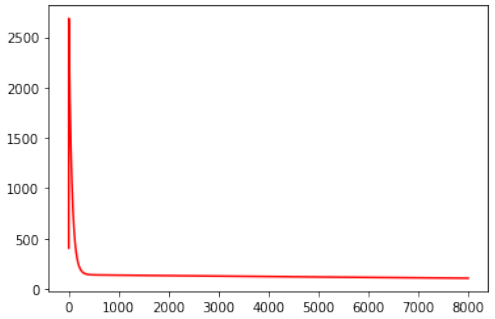


Figura 5: Error con $\alpha = 1{,}0309278350515464e-06$

Podemos ver como la gráfica del error es **monótona decreciente**, lo que indica que el algoritmo encuentra el valor de θ que minimiza el error. Sin embargo, no todos los valores de α proporcionan una buena aproximación, debemos elegir el valor que haga que el error sea mínimo con las iteraciones que hemos realizado. ($\alpha = 0,0001030$).

He realizado 8000 iteraciones del algoritmo para que, al menos, un alpha alcanzase un valor mínimo de error ($\alpha = 0{,}01/97$). El resto de coeficientes de aprendizaje alcanzarán el error mínimo si aumentamos las iteraciones, sin embargo, tardaría demasiado tiempo en ejecutarse.

El algoritmo del método de descenso por gradiente es el siguiente:

```
def descensoPorGradiente(newX,Y,n,m,alpha):

  # Los valores iniciales de theta es un vector con todos sus componentes a \random".
  theta_old = [np.random().rand() for i in range(n)]
  # El vector de thetas nuevo tendrá el mismo tamaño que el antiguo.
  theta_new = [0 for i in range(n)]

  # El criterio de parada que utilizaremos será realizar 8000 iteraciones.
  iteraciones = 8000

  # Para verificar que nuestro algoritmo esta realizando los cálculos de forma
  # adecuada, iremos calculando y almacenando los valores que toma J(theta) en
  # cada una de las iteraciones.
  J = []

  # (repeat until convergence)
  for ejemplos in range(iteraciones):
    error_acumulado = 0
    # Para cada atributo
    for i in range(n):
      # Reiniciamos el sumatorio.
      suma = 0
      # Para cada instancia
      for j in range(m):
        # Calculamos el sumatorio del gradiente del error y
        # lo multiplicamos por alpha
        suma += alpha * ((h(theta_old,newX[j]) - Y[j] )) * newX[j][i]
        error_acumulado += pow(((h(theta_old,newX[j]) - Y[j] )),2)
      # Actualizamos el parámetro de theta restándole a la theta antigua
      # el valor obtenido del sumatorio
      theta_new[i] = theta_new[i] - suma

    # Actualizamos la variable antigua por la nueva.
    theta_old = theta_new

    # Calculamos el error
    J_i =  (1/2) *  error_acumulado
    J.append(J_i)

  return theta_old,J
```

Donde la hipótesis (h) se calcula con la siguiente función:

```
def h(theta,xj):
  return sum([xj[i] * theta[i] for i in range(len(theta))])
```

6. Descenso Por Gradiente Estocástico

Sin embargo, el algoritmo de descenso por gradiente realiza MxN actualizaciones de θ.

La versión estocástica de este algoritmo permite utilizar subconjuntos de instancias aleatorios para que el algoritmo sea más rápido y eficiente, ya que, de esta forma, podemos evitar quedarnos atascados en los mínimos locales.

Para esta versión cogeremos el mismo algoritmo y le añadiremos la capacidad de escoger, aleatoriamente, subconjuntos de las instancias del dataset para ejecutar el código de descenso por gradiente sobre ese conjunto reducido.

Probaremos de nuevo con los mismos coeficientes de aprendizaje divididos entre el número de elementos que tiene cada subconjunto (en este caso 8).

Las rectas de regresión que genera este algoritmo son las siguientes:

- $\alpha = \mathbf{0{,}00125}$: $y = 1{,}19303364 \cdot x - 3{,}89578088$
- $\alpha = \mathbf{0{,}00025}$: $y = 1{,}19243767 \cdot x - 3{,}89025987$
- $\alpha = \mathbf{0{,}000025}$: $y = 0{,}98198715 \cdot x - 1{,}8326068$

Podemos ver como el estocástico necesita menos iteraciones para sacar el error mínimo, ya que con el segundo alpha, de un tamaño de 10 veces menor, se consiguen resultados muy similares al primero.

El código que se ha utilizado para calcular estos coeficientes es el siguiente:

```
def descensoPorGradienteEstocastico(newX,Y,n,m,alpha,batch_size):
  # Los valores iniciales de theta es un vector con todos sus componentes a \random".
  theta_old = [np.random.rand() for i in range(n)]
  theta_new = [0 for i in range(n)]

  iteraciones = 8000

  # Realizaremos la actualización de los parámetros 8000 veces.
  # (repeat until convergence)
  for ejemplos in range(iteraciones):
    # Escogemos un subconjunto de los datos del tamaño del batch
    # que hemos elegido
    batch = [np.random.randint(0,m) for i in range(batch_size)]
    X_select = newX[batch]
    Y_select = Y[batch]

    # Para todos los atributos
    for i in range(n):
      # Reiniciamos el sumatorio.
      suma = 0
      # Para cada instancia
      for j in range(batch_size):
        # Calculamos el sumatorio del gradiente del error y
        # lo multiplicamos por alpha
        suma += alpha * ((h(theta_old,X_select[j]) - Y_select[j] )) * X_select[j][i]
        theta_new[i] = theta_new[i] - suma # Actualización de Thetas

    # Actualizamos la variable antigua por la nueva.
        theta_old = theta_new
  return theta_old
```

He utilizado, también, otros tamaños de *batch* para poder ver como cambian los coeficientes en función de ese parámetro. Los tamaños que he utilizado han sido 4, 8 y 16.

	alpha					
batch_size		0.0125		0.00125		0.000125	
	4	1.19303364	-3.89578083	1.13966713	-3.23892318	0.85896853	-0.57744366
	8	1.19303364	-3.89578088	1.19241095	-3.88989682	0.98346805	-1.8260072
	16	1.19303364	-3.89578088	1.19303364	-3.89578088	1.15845111	-3.55333085

Podemos notar que, para un alpha más grande, el valor de los coeficientes no varía demasiado. Sin embargo, en los alphas más pequeños es necesario escoger un número de ejemplos mayor para que el resultado converja en las 8000 iteraciones que hemos propuesto. En el caso de $\alpha = 0{,}000125$, el coeficiente es tan pequeño que ni cogiendo los 16 ejemplos aleatorios converge en las 8000 iteraciones.

7. Función de Predicción

El objetivo de la regresión lineal es predecir valores que no tengamos en el dataset. Para ello implementaremos una función que, dados los coeficientes y un valor, nos de el resultado correspondiente en la recta.

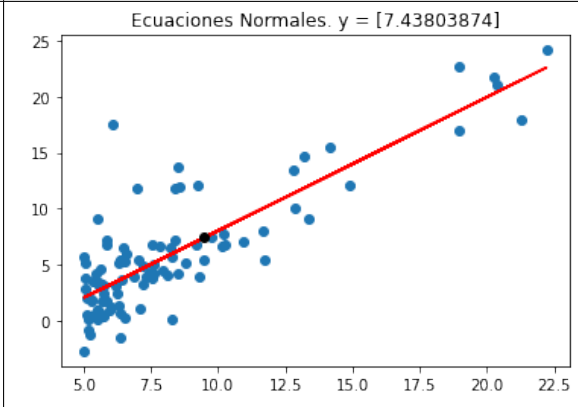
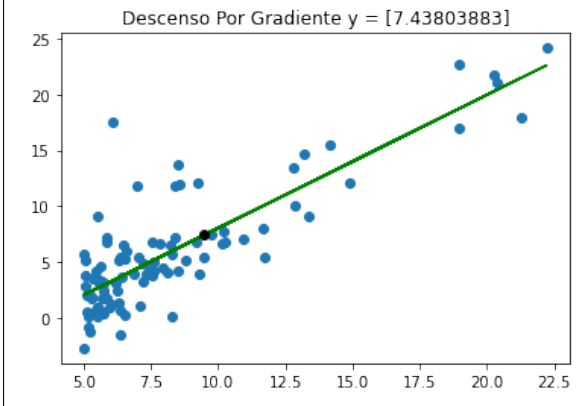
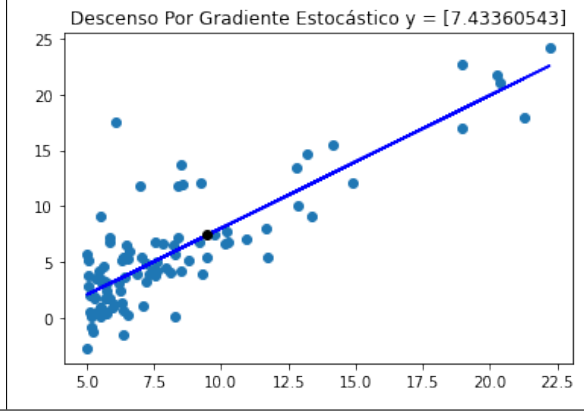
```
def prediccion(thetas,input):
  prediccion = thetas[1]*input + thetas[0]
  return prediccion
```

A la función le pasaremos un vector de coeficientes donde la primera posición será el término independiente y la segunda será la pendiente de la recta.

El último ejercicio de la práctica nos pregunta sobre le beneficio que obtendrá un restaurante con una población estimada de 9.5 mil habitantes. Sabiendo que la X de nuestro problema de regresión son los miles de habitantes de una ciudad y la Y es el beneficio obtenido, podemos utilizar nuestra función de predicción para obtener el resultado. A continuación se muestra una tabla con los tres resultados que proporcionan cada método.

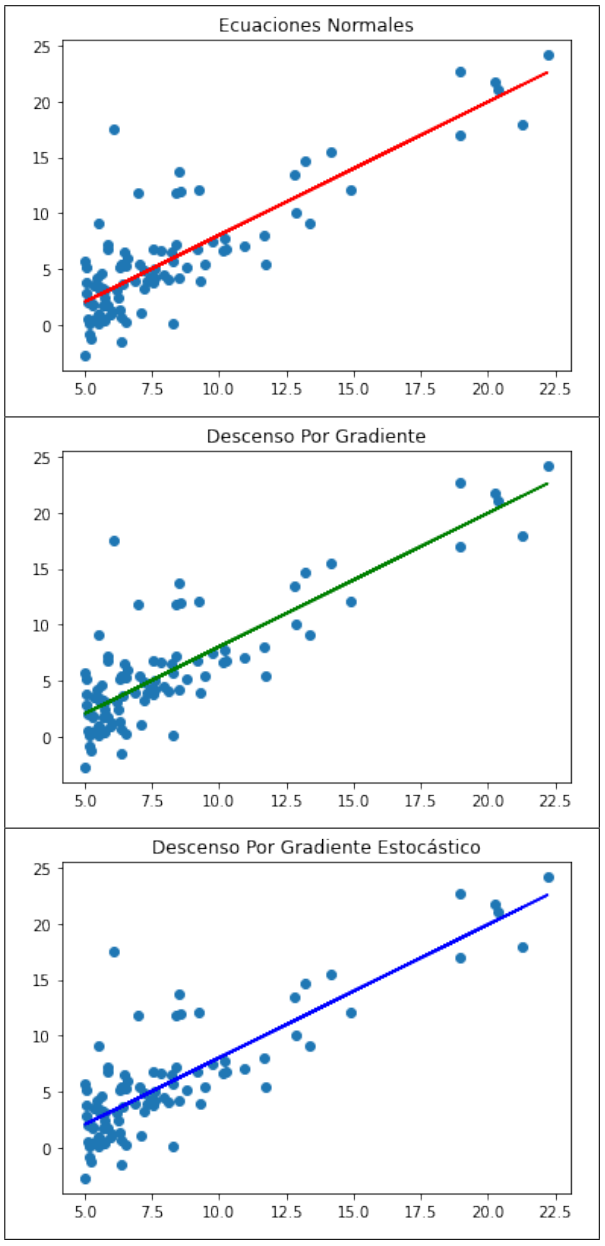
	Ecuaciones Normales	**Descenso por Gradiente**	**Descenso por Gradiente Estocástico**
X = 9,5	Y = 7.43803874	Y = 7.43803874	Y = 7.43360543

Si lo vemos gráficamente obtendríamos lo siguiente:

	X = 9.5
Ecuaciones Normales	
Descenso por Gradiente	
Descenso por Gradiente Estocástico	

8. Comparación de Resultados

En esta sección haremos una comparación entre las diferentes rectas de regresión que hemos obtenido:



Podemos comprobar, gráficamente, como los tres métodos producen resultados aceptables que se ajustan a los datos proporcionados por el fichero de la práctica. Sin embargo, podemos decir que la versión determinista del algoritmo de descenso por gradiente es un poco menos eficiente ya que realiza demasiadas actualizaciones.

En el caso de tener que utilizar la regresión lineal nos decidiríamos por elegir el método analítico de las **ecuaciones normales** y, en caso de que no se pudiera realizar dicho método porque no exista la inversa, utilizaríamos la versión estocástica del descenso por gradiente. Con los dos métodos elegidos tendríamos una regresión lineal rápida y eficaz.