

# Regresión Lineal Simple. Ejemplo minimalista

## Importar las librerías relevantes

```
In [1]: import pandas as pd
import numpy as np
import plotly.express as px
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D # Para graficar en 3-D
```

## Generar datos al azar para entrenar al modelo

Trabajaremos con dos variables de entrada, las  $x_1$  y  $x_2$  en nuestros ejemplos anteriores. Se generan al azar a partir de una distribución uniforme.

Se creará una matriz con estas dos variables. La matriz  $X$  del modelo lineal  $y = x * w + b$

```
In [2]: # Por facilidad, declaramos una variable que indique el tamaño del conjunto
#       de datos de entrenamiento.
observaciones = 1000

x1 = np.random.uniform(low=-10, high=10, size=(observaciones,1))
x2 = np.random.uniform(-10, 10, (observaciones,1))

entradas = np.column_stack((x1,x2))

# Verificar la forma de la matriz
# Debiera ser n x k, donde n es el número de observaciones, y k es el número de v
print (entradas.shape)
```

(1000, 2)

## Generar las metas a las que debemos apuntar

Inventaremos una función  $f(x_1, x_2) = 2 * x_1 - 3 * x_2 + 5 + \text{<ruido pequeño>}$ . El ruido es para hacerlo más realista.

Utilizaremos la metodología de ML, y veremos si el algoritmo la ha aprendido.

```
In [3]: ruido = np.random.uniform(-1, 1, (observaciones,1))

targets = 13 * x1 + 7 * x2 - 12 + ruido

# Veamos Las dimensiones. Deben ser n x m, donde m es el número de variables de s
print (targets.shape)

(1000, 1)
```

## Graficar los datos a usar para el entrenamiento

La idea es ver que haya una fuerte tendencia que nuestro modelo debe aprender a reproducir.

```
In [4]: print(x1.shape)
print(x2.shape)
print(targets.shape)

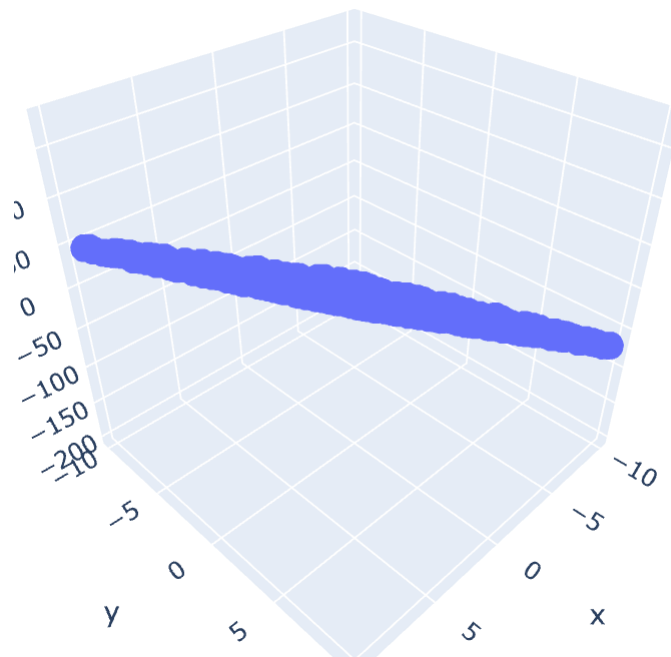
(1000, 1)
(1000, 1)
(1000, 1)
```

```
In [5]: x1N = x1.reshape(observaciones,)
x2N = x2.reshape(observaciones,)
targetsN = targets.reshape(observaciones,)

fig = px.scatter_3d(x = x1N, y = x2N, z = targetsN)

fig.update_layout(
    width = 500,
    height = 500,)

fig.show()
```



## Inicializar variables

Inicializaremos los pesos y sesgos, al azar, dentro de un rango inicial pequeño. Es posible "jugar" con este valor pero no es recomendable ya que el uso de rangos iniciales altos inhibe el aprendizaje por parte del algoritmo


Los pesos son de dimensiones  $k \times m$ , donde  $k$  es el número de variables de entrada y  $m$  es el número de variables de salida.

Como solo hay una salida, el sesgo es de tamaño 1, y es un escalar

```
In [6]: rango_inicial = 0.1

weights = np.random.uniform(low = -rango_inicial, high = rango_inicial, size=(2,
biases = np.random.uniform(low = -rango_inicial, high = rango_inicial, size=1)

#Veamos cómo fueron inicializados.
print (weights)
print (biases)
```



```
[[ -0.02105953]
 [  0.02619776]]
[-0.09741205]
```

```
In [7]: weights.shape
```

```
Out[7]: (2, 1)
```

## Asignar la tasa de aprendizaje (Eta)

Se asigna un a tasa de aprendizaje pequeña. Para este ejemplo funciona bien 0.02. Vale la pena "jugar" con este valor para ver los resultados de hacerlo.

```
In [8]: eta = 0.001
```

## Entrenar el modelo

Usaremos un valor de 100 para iterar sobre el conjunto de datos de entrenamiento. Ese valor funciona bastante bien con la tasa de aprendizaje de 0.02. Cómo saber el número adecuado de iteraciones es algo que veremos en futuras sesiones, pero generalmente una tasa de aprendizaje baja requiere de más iteraciones que una más alta. Sin embargo hay que tener en mente que una tasa de aprendizaje alta puede causar que la pérdida "Loss" diverja a infinito, en vez de converger a cero (0)

Usaremos la función de pérdida L2-norm, pero dividido por 2, Para se consistente con la clase. Es más, también lo dividiremos por el número de observaciones para obtener un promedio de pérdida por observación. Hablamos en clase sobre la posibilidad de modificar esta función una vez no se pierda la característica de ser más baja para los resultados mejores, y vice versa.

Imprimimos la función de pérdida (loss) en cada iteración, para ver si está decreciendo como se desea.

Otro pequeño truco es escalar las deltas de la misma manera que se hizo con la función de pérdida. De esta forma la tasa de aprendizaje es independiente del número de muestras (samples u observaciones). De nuevo esto no cambia el principio, solo hace más fácil la selección de una tasa única de aprendizaje.

Finalmente aplicamos la regla de actualización del decenso de gradiente.

Ojo! los pesos son 2x1, la tasa de aprendizaje es 1x1 (escalar), las entradas son 1000x2, y las deltas escaladas son 1000x1. Necesitamos obtener la transpuesta de las entradas para que no hayan problemas de dimensión en las operaciones.

```
In [9]: for i in range (100):

    # Esta es la ecuacion del modelo lineal:  $y = xw + b$ 
    salidas = np.dot(entradas, weights) + biases

    # Las deltas son las diferencias entre las salidas y las metas (targets)
    # deltas es un vector 1000 x 1
    deltas = salidas - targets

    loss = np.sum(deltas ** 2) / 2 / observaciones

    print(loss)

    deltas_escaladas = deltas / observaciones

    weights = weights - eta * np.dot(entradas.T, deltas_escaladas)
    biases = biases - eta * np.sum(deltas_escaladas)

    # Los pesos son actualizados en una forma de algebra lineal (una matriz menos
    # Sin embargo, los sesgos en este caso son solo un número, es necesario trans
    # a un escalar.
    # Ambas lineas son consistentes con la metodología de descenso de gradiente-
```

829.559752385146  
779.3460139439856  
732.4252853826832  
688.5810626264001  
647.6110771747271  
609.3263600767206  
573.5503674523051  
540.1181635131687  
508.8756573023715  
479.6788896204814  
452.3933668383214  
426.89343851337526  
403.0617159296265  
380.7885288699868  
359.9714181073999  
340.5146612660107  
322.32882985821357  
305.3303754476735  
289.44124302319756  
274.5885007042655

**Desplegamos los pesos y sesgos para ver si funcionaron correctamente.**

Por el diseño de nuestro datos, los pesos debieran ser 2 y -3, y el sesgo: 5

**NOTA:** Si aún no están los valores correctos, puede que aún estén convergiendo y sea necesario iterar más veces. Para esto solo se requiere ejecutar la celda anterior cuantas veces sea requerido

```
In [10]: print(weights, biases)
```

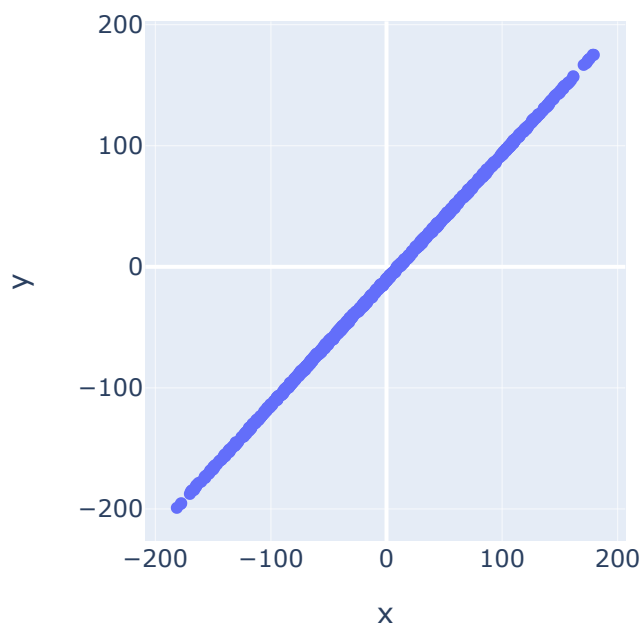
```
[[12.5918193 ]  
 [ 6.69043016]] [-1.21459488]
```

## Graficar las últimas salidas vrs las metas (targets)

Como son las últimas, luego del entrenamiento, representan el modelo final de exactitud. Enntre más cercana esté esta gráfica a una línea de 45 grados, mhás cercanas están las salidas y metas.

Como este ejemplo es pequeño, es posible hacerlo, en los problemas que veremos más tarde en la clase, esto ya no sería posible.

```
In [11]: salidasN = salidas.reshape(observaciones,  
targetaN = targets.reshape(observaciones,  
fig = px.scatter(x = salidasN, y = targetaN)  
  
fig.update_layout(  
    width = 400,  
    height = 400,  
  
fig.show()
```



```
In [ ]:
```

