

Regresión

Cómo vimos en la sección de *Machine Learning* la regresión es un **método matemático** utilizado para inferir un valor numérico en base a datos observados de ocurrencia de un fenómeno.

Por ejemplo, un fenómeno sencillo sería observar cuanta energía se le suministra a un fluido líquido hasta cambiar de estado de agregación. Otro podría ser observar que a medida que un cohete que sale desde la tierra, al aumentar la elevación disminuye la temperatura atmosférica hasta alcanzar la troposfera.

Es decir relacionar **características continuas** con **etiquetas continuas** (temperatura [C°] en función de la altura[m]).

Otro ejemplo: si queremos determinar cuál es la probabilidad que existe de que un usuario vuelva a un sitio web, esto puede estar basado en: si queremos predecir el **número** de clics en el futuro, o qué **cantidad** de impresiones de un anuncio tendremos. En este caso estamos en un problema de regresión.

Modelos de Regresión

Los modelos de regresión sirven para predecir una variable cuantitativa. Es decir, nos interesa obtener números.

Sin embargo, estos número tendrán sentido si existe **correlación** entre la variable regresora (x_1, x_2, \dots, x_n) y la explicada (y).

Por correlación entendemos una relación entre variables de tal forma que si una aumenta o disminuye la otra se ve afecta en forma directa o inversa a dichas variaciones.

Un análisis de **correlación** se puede hacer con el estadístico **r (coeficiente de correlación de Pearson)** el cual va a tomar valores entre [-1,0,1] siendo una correlación inversa o directa, y si el coeficiente es aproximadamente cero no existe correlación.

En base a esto podemos hablar de tipos de modelos de regresión:

- Lineal:
 - Simple
 - Multivariante
- No lineal:
 - Polinomial
 - Logarítmica

X : regresora, predictora, explicativa, **feature**, característica, variable independiente.

Y: respuesta, explicada, objetivo, **target**, **label**, etiqueta, variable dependiente.

Aprendizaje supervisado —> $f(\text{features}) = \text{labels}$

Regresión Lineal Simple

La regresión lineal simple hace honor a su nombre: es un enfoque muy sencillo para predecir una respuesta cuantitativa Y sobre la base de una única variable regresora X.

Se supone que existe una relación aproximadamente lineal entre X e Y.

$$Y = \beta_0 + \beta_1 X$$

En la ecuación, β_0 y β_1 son dos constantes desconocidas que representan los términos **intercepto** y **pendiente** del modelo lineal. Ambos términos se conocen como **coeficiente o parámetros** del modelo.

Lo que hacemos cuando **entrenamos** un modelo lineal, es averiguar que valores de **parámetros** b_0 y b_1 que se aproximen a los valores reales de la relación lineal.

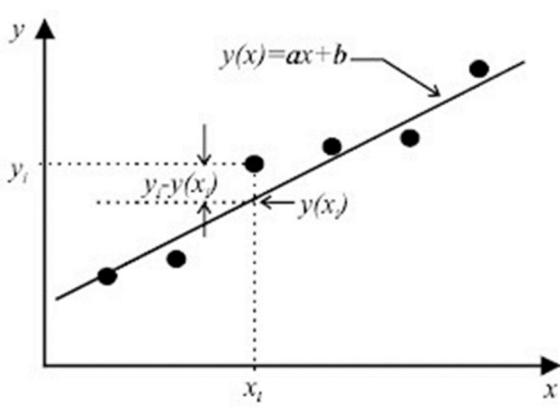
$$\hat{y}_i = b_0 + b_1 x_i$$

Pero, ¿cómo hacemos para conocer aquellos parámetros que mejor se aproximan a β_0 y β_1 ? Esto lo haremos con el método de **OLS** o mínimos cuadrados ordinarios.

Ordinary Least Square

Este método fue creado en 1795 por Karl Gauss (a los 18 años).

Para esto primero debemos saber cómo medimos el error entre el valor predicho y el valor original. Si tuviéramos un conjunto de puntos como vemos en la gráfica, podríamos resolver el error por OLS de la siguiente manera



Estimado los valores de parámetros b_0 y b_1 (en la gráfica serían b y a respectivamente) podemos establecer una recta promedio que para cada valor de X_i nos de un Y_i bastante acertado. Pero esto se logra mediante muchas iteraciones. Primero se proponen dos valores de b_0 y b_1 se calcula el error para cada valor X_i —> $(b_0 + b_1 X_i) - Y_i = y(x_i) - Y_i$

Entonces, el método de OLS nos permite conocer los parámetros de la relación lineal simple, mediante la sumatoria de los **residuos al cuadrado (RSS)**, que son las

diferencias entre Y_i e $\hat{Y}(x_i)$.

$$e_i = y_i - \hat{y}_i$$

$$\sum_{i=1}^n e_i^2 = \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \sum_{i=1}^n (y_i - b_0 - b_1 x_i)^2$$

Ecuación del método de OLS.

El método OLS, buscará minimizar el **RSS**, eligiendo valores de b_0 y b_1 que mejor modelen la relación, es decir aproximándose a β_0 y β_1 .

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} \quad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

Diagnóstico del modelo (Estadísticos)

Supuesto de linealidad: si el modelo es correcto, los residuos se distribuyen linealmente, con una distribución normal cuya media es cero. Contrariamente, si los residuos están sesgados, es decir se distribuyen de forma *no lineal*, el modelo estará incorrectamente especificado.

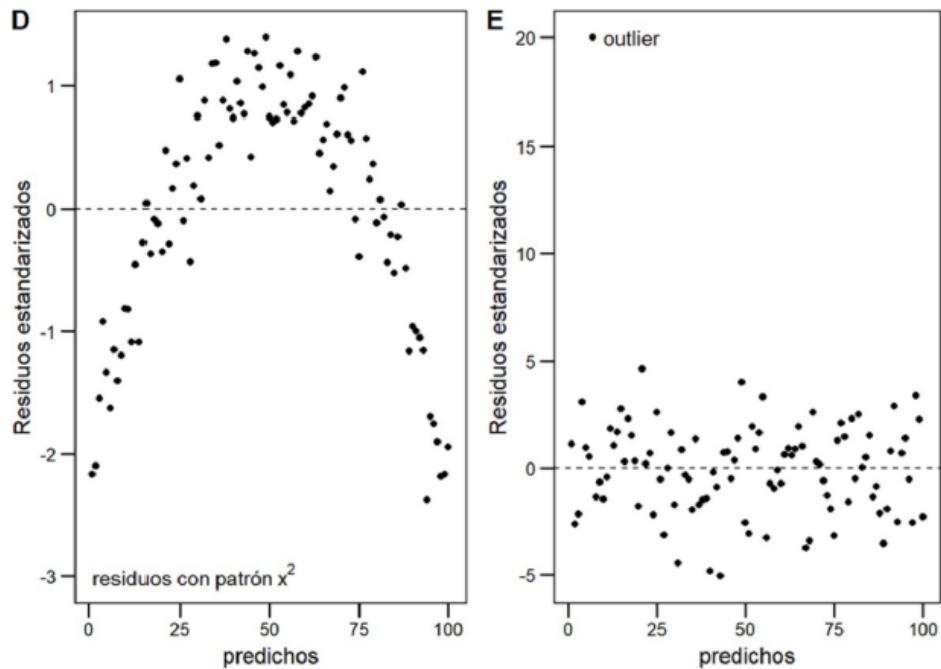
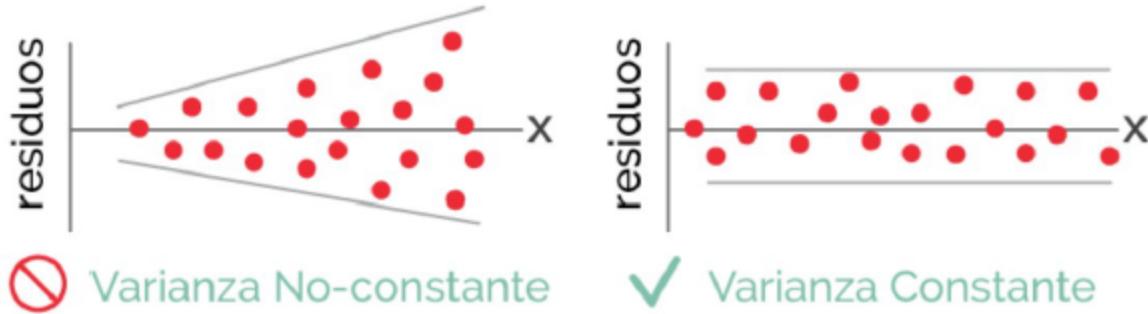
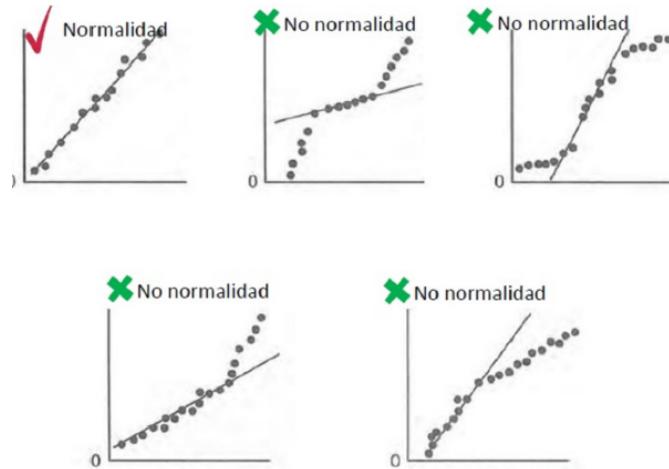


Imagen que muestra: (izq) un conjunto de datos de los residuos estandarizados que igualmente están sesgados sistemáticamente, (der) conjunto de datos de los residuos estandarizados con distribución lineal.

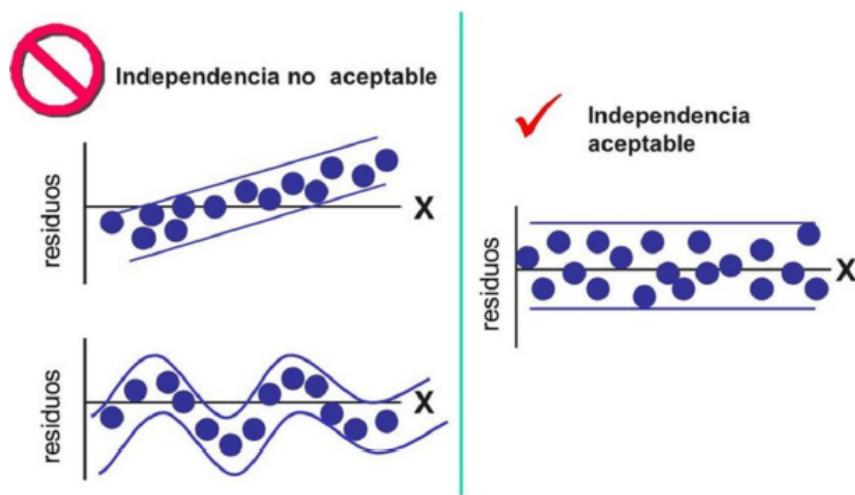
Supuesto de homocedasticidad: se aceptará este supuesto si los residuos se distribuyen de forma constante. Veremos una nube de puntos de forma similar en todo el rango de observaciones de la variable independiente.



Supuesto de normalidad: una forma sencilla es representar gráficamente los residuos, en los que deberíamos ver su distribución lineal.



Supuesto de independencia: los residuos deben ser independientes entre sí y no debe haber ningún tipo de correlación entre ellos.



En caso de no superar dichos supuestos, la alternativa es hacer un buen análisis descriptivo, tener en cuenta si no hay una variable que se pasa por alto, analizar correlación y elegir otro modelo que mejor represente dichos datos (tal vez polinómica).

Supuesto de colinealidad: esto es para regresión multivariante (no aplica para lineal simple)

Lasso regresión (L1 regularización)

Este método de regresión **penalizado** se usa a menudo para seleccionar un subconjunto de variables, ya que, impone una **restricción** a la suma del sesgo absoluto de los parámetros del modelo. Esta restricción hace que los coeficientes de

regresión para algunas variables se reduzcan a cero. Esto se logra aplicando un parámetro de ajuste **Lambda** al modelo para controlar la fuerza de la penalización. A medida que aumenta lambda, más coeficientes se reducen a cero. Cuando lambda es igual a cero, entonces tenemos una regresión **OLS**, el sesgo aumenta y la varianza disminuye a medida que aumenta lambda.

$$\sum_{i=1}^n (y_i - \sum_j x_{ij} \beta_j)^2 + \lambda \sum_{j=1}^p |\beta_j|$$

Lasso es importante porque puede proporcionar una mayor precisión de predicción sobre la verdadera relación entre la variable de respuesta y los predictores. También porque en el modelo de regresión de *Mínimos Cuadrados Ordinarios (OLS)*, podemos observar un sesgo bajo y una varianza baja; sin embargo, si tiene un número relativamente pequeño de observaciones y una gran cantidad de predictores, la varianza de los parámetros de OLS será mayor. Con la regresión de Lasso, la reducción de los coeficientes puede reducir la varianza sin un aumento sustancial del sesgo. Además, la regresión de Lasso puede aumentar la interpretabilidad del modelo. A menudo, algunas variables explicativas en la regresión lineal múltiple no están realmente asociadas con la variable de respuesta y terminaríamos con *overfitting* y difícil de interpretar. Con Lasso los coeficientes de las variables sin importancia se reducen a cero.

Ridge regresión (L2 regularización)

Ridge es una técnica muy popular para mejorar la exactitud de las predicciones. Mejora los errores de predicción al reducir en tamaño los coeficientes de regresión que sean demasiado grandes para reducir el *overfitting*, pero no realiza selección de variables y por tanto no produce un modelo más interpretable, como es el caso de *Lasso*.

$$\sum_{i=1}^n (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^P \beta_j^2$$

Regresión Multivariante

Como vimos anteriormente, en una regresión, si la variable **explicativa** es una sola X, estamos frente a un modelo de regresión lineal simple. En cambio si hay más de una nuestro modelo es de **regresión lineal múltiple**.

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_n X_n$$

Nótese que ahora hay más de una variables explicativas, sin embargo, se mantiene el concepto de **linealidad** ya que cada variable explicativa es independiente entre ellas. No existe relaciones dependientes de otra variable, como si puede ocurrir en una función polinómica de segundo grado.

Cada vez que agregamos más variables **creamos nuevos modelos**.

Criterio para elegir el Modelo

La idea de la regresión es construir un modelo que sea representativo un conjunto de datos (X) de una población y poder inferir sobre ésta misma algún fenómeno (y).

Anteriormente dijimos que cada variable que se agrega es un modelo nuevo. Pero, **¿hasta cuando es bueno agregar variables?**

Al tener la posibilidad de incluir más variables, se nos abre un campo casi infinito de combinaciones:

1. Podemos agregar variables a las originales.
2. Podemos agregar nuevas variables.
3. Podemos sacar variables antiguas.
4. Transformar variables.

5. Sumar variables.

Entonces, para conocer cual es el mejor modelo de estas muchas variantes tenemos **métodos y criterios**.

- **Criterios Directos:** validación simple o cruzada.
- **Criterios Indirectos:** R2 adj, AIC, BIC.

Criterios Directos

- Validación Simple: Esto lo vimos en teoría de Machine Learning. Consiste en preparar el dataset (conjunto de datos) dividiéndolo en un 70% para *entrenar*, un 15% para *validar* y un 15% para realizar un *test* final al modelo. Este método es el más simple **pero** no permite utilizar los datos de validación para entrenar y viceversa. Tiene una separación fija y estructurada de los datos. Este método es **directo** porque nos garantiza lograr un **buen** (buenas métricas) modelo (aunque no sea el mejor).
- Validación Cruzada: Este método es el más utilizado, puesto que permite utilizar el conjunto de validación para entrenar y viceversa. Requiere más capacidad de procesamiento pero nos permite utilizar todos los datos, excepto el conjunto de test. Consiste en elegir un parámetro **k**, que puede ir de [5,10] que divide los datos en entrenamiento y validación, y durante **k** iteraciones, irá agrupando los datos de validación y entrenamiento, de diferentes formas.

Criterios Indirectos

- R2 adj: el coeficiente r cuadrado ajustado, es una métrica, se calcula a partir del modelo ya entrenado, por esto es que es indirecto. Cuanto mayor sea, mejor será el poder de explicación del modelo (ejemplo: si el R2adj=0.7 significa que las variables elegidas explican en un 70% al modelo).
- AIC (criterio de información Akaike): es un indicador, basado en teoría de la información, que nos muestra la **calidad relativa de un modelo**, dado un conjunto de datos. Lo que intenta es elegir el modelo que minimice la **perdida de la información**. Para esto utiliza una función de *máxima verosimilitud* donde se tiene en cuenta los modelos posibles y sus parámetros, y elegirá aquel conjunto de parámetros que sean más verosímil.

- BIC (criterio de información bayesiano): es un método de selección de modelos basado en una *función de probabilidad* que se demuestra a partir de **Bayes**, siendo inversa a la función de *verosimilitud*, por lo que este criterio está relacionado con el **AIC**. Funciona de forma parecida, midiendo la pérdida de información, y en base a esto eligiendo el mejor modelo para un conjunto de datos.

Criterio para elegir las Variables

Al momento de elegir variables, siempre se va a cumplir el **Principio de Parsimonia**: un modelo parsimonioso (menor número de variables) proporciona mejores predicciones que un modelo completo. Esto está relacionado con el problema del **sobreajuste (overfitting)**. Los modelos con muchas variables pueden “memorizar” los datos, perdiendo poder de “generalización” frente a datos nuevos. A la vez que siempre se van a necesitar muchas variables para realizar una predicción.

Por lo que ser cuidadosos y criteriosos al momento de agregar variables es importante, para esto existen diversos métodos.

Sub-set selection: consiste en crear todos los modelos posibles, usando las combinaciones posibles entre variables. Tal vez para un conjunto de datos con dos o tres variables independientes resulta simple de lograr. Pero cuando se tienen muchas variables es *inviable* semejante esfuerzo de combinaciones.

- La combinación de variables siempre es exponencial: 2^nX
- Es muy propenso a lograr overfitting.

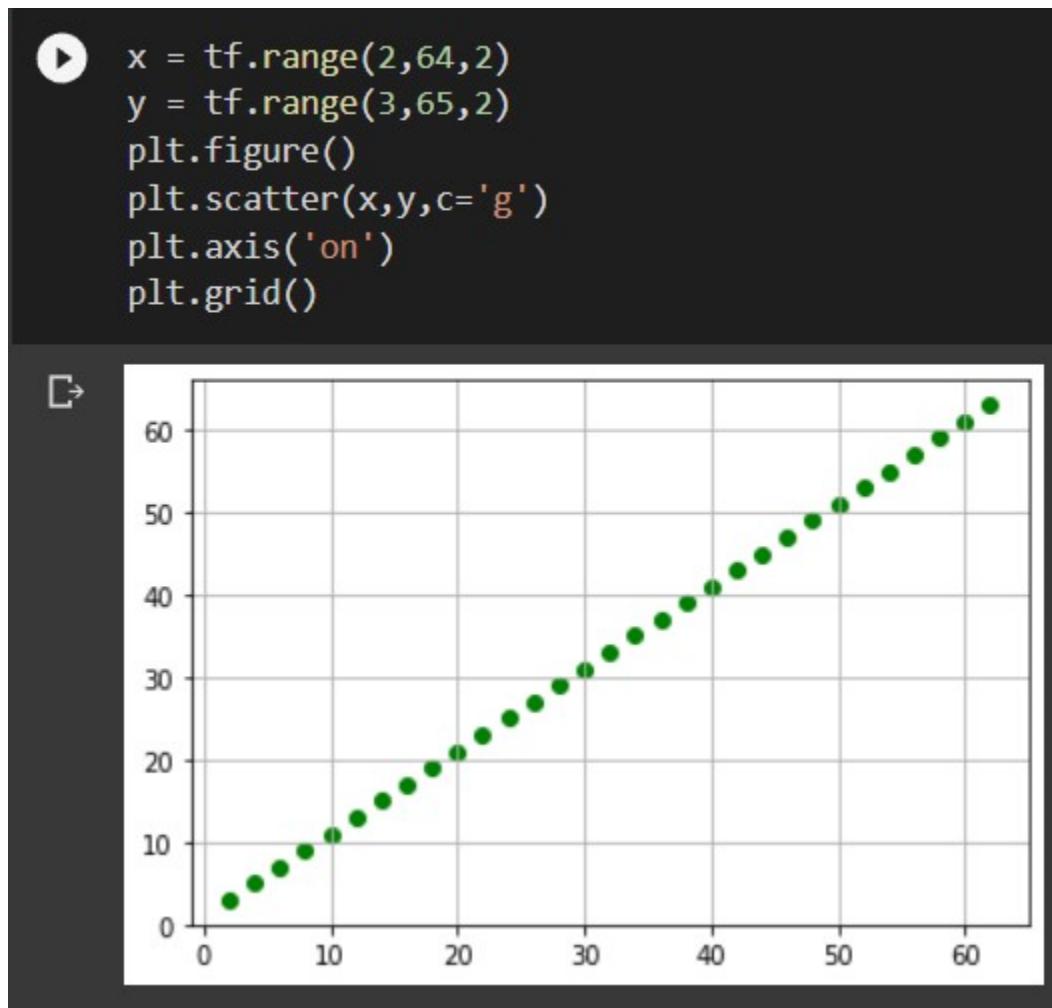
Forward Selection: consiste en comenzar a probar con un modelo que tenga solo una variable e ir monitoreando el **R2adj**, a medida que se agregan variables buscar el punto en el que la métrica disminuya, entonces el modelo anterior es el correcto.

Backward Selection: comienza con todos los predictores (X_i) y luego se van probando todos los modelos sacando de a una variable, hasta que el criterio seleccionado deje de mejorar ($R2adj$).

Stepwise: este método es una combinación de *forward selection* y *backward selection*. Es el método más utilizado ya que permite llegar a un buen modelo sin requerir tanto cálculo.

Ejemplos prácticos

Tal vez para el lector al observar un conjunto de datos como el siguiente le parezca simple darse cuenta de la relación que existe entre la variable independiente y la dependiente, pudiendo concluir la misma sin necesidad de Machine Learning.



Si puede observar el lector a simple vista que para cada valor en x es casi el mismo en y pero con una pequeña diferencia. Esta se puede observar mejor en detalle con los valores que componen a x e y.

```
[21] x,y
```

```
(<tf.Tensor: shape=(31,), dtype=int32, numpy=
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34,
       36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62],  
      dtype=int32)>, <tf.Tensor: shape=(31,), dtype=int32, numpy=
array([ 3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35,
       37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63],  
      dtype=int32)>)
```

Ambos tensores son iguales, son vectores de dimensión (31,) es decir rango 1 y los valores se corresponden de la siguiente forma: ($x[0] = 2, y[0]=3$), ($x[1]=4, y[1]=5$), ($x[2]=6, y[2]=7$), etc; esta relación resulta notoria de ser:

$$y = x + 1$$

Esto mismo podemos dejar que concluyan los algoritmos de Machine Learning para este ejercicio.

Para esto utilizaremos las clases *LinearRegression*, *Lasso* y *Ridge* de la librería *Scikit-Learn*, las cuales son clases que instanciaremos para calcular esa relación de forma automática.

Luego utilizaremos redes neuronales en un ejercicio final para comparar la performance que es posible obtener con un algoritmo de Machine Learning instanciado desde *Scikit-Learn* versus una red neuronal sencilla construida mediante *stacking* de capas y neuronas.

Entrenamiento y Testeo

Para esta tarea se divide los datos de este pequeño ejemplo en porcentajes de la siguiente manera:

- Train: 80% del total de datos.
- Testeo: 20% del total de datos.

De manera tal de poder entrenar los modelos con el conjunto de datos de entrenamiento de forma supervisada, y luego con los datos de testeo poner a prueba los

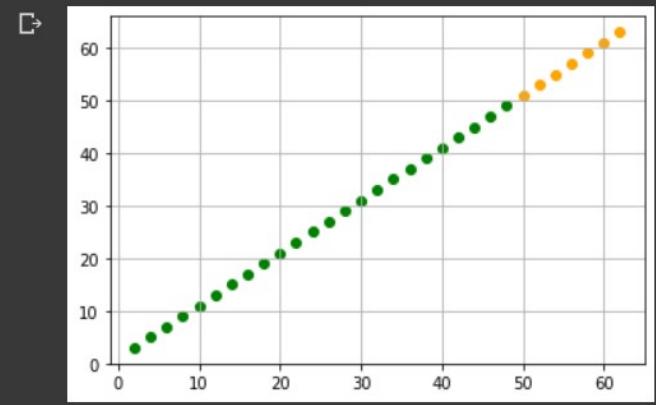
modelos tratando de inferir correctamente sus etiquetas.

Cabe destacar que en este ejercicio no se utilizó un conjunto de validación debido a la sencillez y escasez de datos. Es considerable en problemas del mundo real aplicar una división considerando la validación ya que es la primera prueba que recibe el modelo después de entrenarse para poder ajustar cualquier parámetro antes de la prueba real que se realiza con el conjunto de testeo.

```
[22] split_size = int(31*0.8)
      x_train,y_train = x[:split_size],y[:split_size]
      x_test,y_test = x[split_size:],y[split_size:]
      x_train.shape,x_test.shape
```

```
(TensorShape([24]), TensorShape([7]))
```

```
▶ plt.figure()
  plt.scatter(x_train,y_train,c='g')
  plt.scatter(x_test,y_test,c='orange')
  plt.axis('on')
  plt.grid()
```



Regresión Lineal

```
[43] #Linear Regression
      from sklearn.linear_model import LinearRegression
      lr = LinearRegression()
      lr.fit(x_train.reshape(-1, 1),y_train.reshape(-1, 1))
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

```
▶ lr.score(x_test.reshape(-1, 1),y_test.reshape(-1, 1))
```

```
↪ 1.0
```

```
[48] lr.predict(x_test.reshape(-1,1)).mean(),y_test.mean()
```

```
(57.0, 57.0)
```

Podemos observar que el entrenamiento del modelo tiene un 1.0 en el *score* con el conjunto de testeo. Esto quiere decir que predice con una precisión del 100% cada uno de los datos de ese conjunto. Podemos observar que la media (*mean()*) de las predicciones es igual a la media de las etiquetas de ese mismo conjunto.

Pero que sucede si le pasáramos datos nuevos que se salen de los valores de entrenamiento y testeo.

Vemos que la relación $y = x + 1$ en el algoritmo está calculada a la perfección, permitiéndonos así poder generalizar con nuevos datos

```
▶ sample = np.array([128,200,250])
  lr.predict(sample.reshape(-1,1))
→ array([[129.],
       [201.],
       [251.]])
```

Regresión Lasso

```
[12] #Lasso Regression
    from sklearn.linear_model import Lasso
    la = Lasso()
    la.fit(x_train[:,np.newaxis],y_train[:,np.newaxis])

    Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
          normalize=False, positive=False, precompute=False, random_state=None,
          selection='cyclic', tol=0.0001, warm_start=False)

[13] la.score(x_test[:,np.newaxis],y_test[:,np.newaxis])
    0.9983378071833648

[15] la.predict(x_test[:,np.newaxis]).mean(),y_test.mean()
    (56.83826086956522, 57.0)
```

Podemos observar que la implementación de este modelo de regresión lineal *Lasso* obtiene valores óptimos para *accuracy* en el conjunto de datos de testeo, así como también, el promedio de las predicciones es un valor muy aproximado al promedio de las etiquetas originales.

Regression Ridge

```

▶ from sklearn.linear_model import Ridge
ri = Ridge()
ri.fit(x_train[:,np.newaxis],y_train[:,np.newaxis])

↳ Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
         normalize=False, random_state=None, solver='auto', tol=0.001)

[21] ri.score(x_train[:,np.newaxis],y_train[:,np.newaxis])
0.999999527615615

[22] ri.score(x_test[:,np.newaxis],y_test[:,np.newaxis])
0.9999971155028478

▶ ri.predict(x_test[:,np.newaxis]).mean(),y_test.mean()
(56.993262334275165, 57.0)

```

Según se puede observar el modelo Ridge posee valores con mayor confianza en el *accuracy* del conjunto de testeo y en el promedio de las predicciones con respecto al promedio de las etiquetas originales. Esto nos indica que Ridge es un algoritmo muy exacto sin caer en *overfitting* como en el caso de *Regression Linear*.

Ridge es una técnica muy popular para mejorar la exactitud de las predicciones. Mejora los errores de predicción al reducir en tamaño los coeficientes de regresión que sean demasiado grandes para reducir el *overfitting*, pero no realiza selección de variables y por tanto no produce un modelo más interpretable, cómo es el caso de *Lasso*.

Aplicación

Un ejemplo de aplicación podría ser tal vez determinar la relación entre un conjunto de datos observados cómo puede ser la escala para medir temperatura de Anders Celsius (escala centígrada) y la escala graduada por Daniel Gabriel Fahrenheit, escala que va desde 32 grados hasta 212 grados.

Si bien es conocida la relación matemática que permite calcular la temperatura en una escala conociendo el valor en la otra escala, sería más interesante posibilitar esta tarea mediante el uso de Machine Learning.

Para esto usaremos un método supervisado de entrenamiento donde los datos de entrada sean los valores continuos de la escala de Celsius (0-100) °C y los datos de salida los valores en la escala de Fahrenheit.

```
▶ #Ejercicio de aplicación
import numpy as np
x = np.arange(0,100,1,dtype=int)
y = (x * 9/5) + 32
y = np.cast['int32'](y)
print(f"Los valores en Celsius: {x},\n Los valores en Fahrenheit: {y}")

↳ Los valores en Celsius: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99],
Los valores en Fahrenheit: [ 32  33  35  37  39  41  42  44  46  48  50  51  53  55  57  59  60  62
 64  66  68  69  71  73  75  77  78  80  82  84  86  87  89  91  93  95
 96  98 100 102 104 105 107 109 111 113 114 116 118 120 122 123 125 127
129 131 132 134 136 138 140 141 143 145 147 149 150 152 154 156 158 159
161 163 165 167 168 170 172 174 176 177 179 181 183 185 186 188 190 192
194 195 197 199 201 203 204 206 208 210]
```

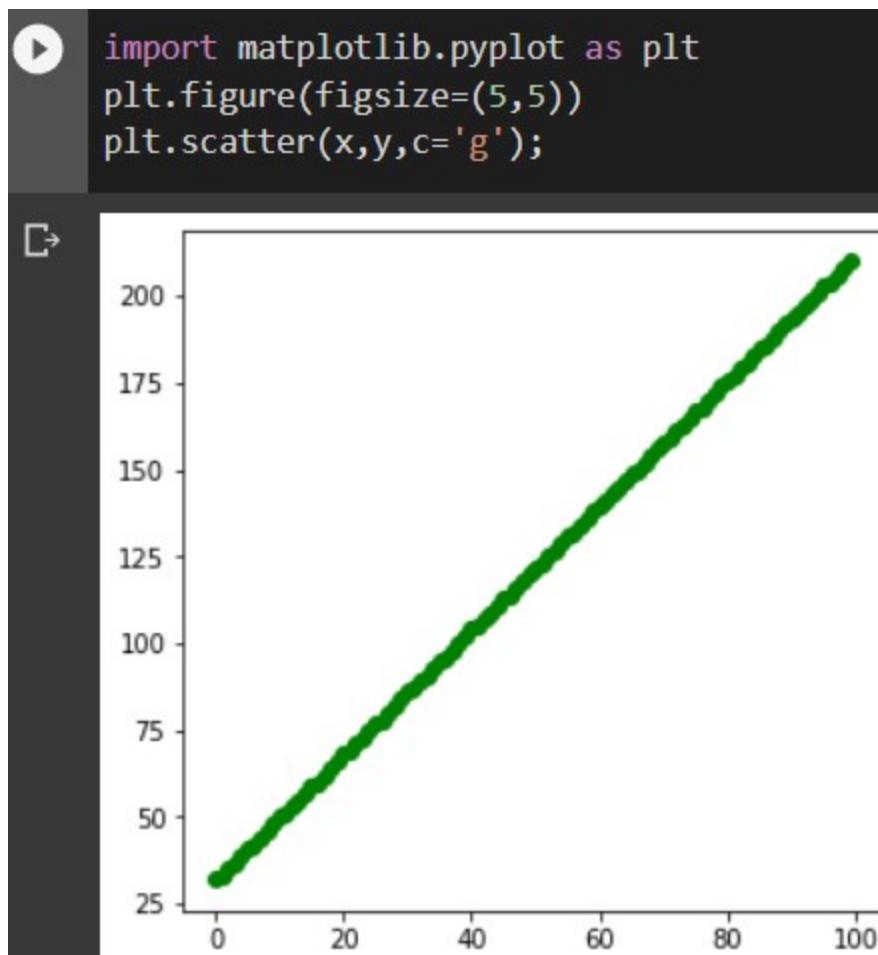


Figura que muestra la relación lineal que existe entre ambas escalas.

Con esto creamos nuestro dataset donde x son los valores de la escala Celsius e y son los valores de la escala Fahrenheit según la relación matemática. Esta misma relación es la que dejaremos que el algoritmo aprenda por si solo tanto la *pendiente* como el *bias* de tal manera que nos aseguren que al ingresar un valor en Celsius podemos obtener a la salida un valor en escala Fahrenheit.

Esto lo realizaremos con una Regresión del tipo Lasso y lo compararemos con una red neuronal.

Lasso

Primer paso es dividir el conjunto de datos en datos de entrenamiento y de testeo. Esto lo haremos de manera

```

[37] split_size = int(len(x)*0.8)
      x_train, x_test = x[:split_size], x[split_size:]
      y_train, y_test = y[:split_size], y[split_size:]
      x_train.shape,y_train.shape,x_test.shape,y_test.shape

```

sencilla, más adelante veremos cómo hacerlo con **Scikit-Learn**.

Luego implementamos *Lasso* instanciando la clase respectiva de *Scikit-Learn*. Podemos observar que al correr el método *fit()* nos muestra aquellos atributos que son públicos y mediante los cuales podemos parametrizar el entrenamiento. Es destacable que *Scikit-Learn* ofrece por defecto un entrenamiento con la mejor combinación de hyperparametros, los cuales están probados y calibrados para la generalidad de caso; aunque no se descarta la posibilidad de ajustarlos de forma manual.

```
[38] #Lasso Regression
      from sklearn.linear_model import Lasso
      la = Lasso()
      la.fit(x_train[:,np.newaxis],y_train[:,np.newaxis])

      Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
            normalize=False, positive=False, precompute=False, random_state=None,
            selection='cyclic', tol=0.0001, warm_start=False)

▶ la.score(x_test[:,np.newaxis],y_test[:,np.newaxis])
⇒ 0.9991753200927999

[40] la.predict(x_test[:,np.newaxis]).mean(),y_test.mean()

(192.60623534927333, 192.7)
```

Podemos observar que el entrenamiento es satisfactorio pudiendo generalizar con un alto valor de *accuracy* para el conjunto de testeo. También se puede observar que el valor promedio de las predicciones con el conjunto de testeo es prácticamente el mismo valor promedio de las etiquetas originales.

Red neuronal

Para este algoritmo de Machine Learning usaremos el mismo conjunto subdividido anteriormente en entrenamiento y testeo.

Se arma la estructura de la red mediante el método de apilar (*stacking*) capas profundas con determinada cantidad de neuronas cada una. Para esto usaremos la API de alto nivel de *TensorFlow* que es el módulo *Keras*. De este módulo usaremos el

método Sequential para construir la red neuronal que para este ejercicio resulta sencilla (*shallow*) ya que no necesita mucho poder de cálculo para esta tarea.

```
#Red neuronal
import tensorflow as tf
modelo = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(1,)),
    tf.keras.layers.Dense(4),
    tf.keras.layers.Dense(4),
    tf.keras.layers.Dense(1,activation='linear')
])
modelo.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_9 (Dense)	(None, 4)	8
dense_10 (Dense)	(None, 4)	20
dense_11 (Dense)	(None, 1)	5
Total params: 33		
Trainable params: 33		
Non-trainable params: 0		

Podemos observar que la red se compone de la capa de entrada sin parámetros, y luego le siguen dos capas ocultas con 4 neuronas (*nodos*) cada una para terminar en la capa de salida con una única neurona que nos devuelve un valor en función del valor de entrada y los parámetros de la función de activación.

Luego compilamos el modelo. Este paso resulta en configurar los hyperparametros de lo que será el entrenamiento, es decir, cómo se calculará el error de la función de costos entre los valores de salida y los originales, cómo será optimizado el aprendizaje, entre otras cosas (para mayor entendimiento visitar la documentación de [Keras](#)).

```
modelo.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
               loss=tf.keras.losses.mae,
               metrics=['mae'])
```

Finalmente, podemos entrenar el modelo con el método *fit()*, el mismo es parametrizado con los datos de entrenamiento, de testeo, la cantidad de épocas, entre otros argumentos (para mayor información visitar la documentación de [TensorFlow](#)).

```
modelo.fit(x_train,y_train,
            epochs=50,
            validation_data=(x_test,y_test),
            workers=8)
```

```
Epoch 44/50
3/3 [=====] - 0s 17ms/step - loss: 0.2761 - mae: 0.2761 - val_loss: 0.3550 - val_mae: 0.3550
Epoch 45/50
3/3 [=====] - 0s 14ms/step - loss: 0.2994 - mae: 0.2994 - val_loss: 0.3638 - val_mae: 0.3638
Epoch 46/50
3/3 [=====] - 0s 14ms/step - loss: 0.2893 - mae: 0.2893 - val_loss: 0.5689 - val_mae: 0.5689
Epoch 47/50
3/3 [=====] - 0s 14ms/step - loss: 0.2905 - mae: 0.2905 - val_loss: 0.2734 - val_mae: 0.2734
Epoch 47/50
3/3 [=====] - 0s 17ms/step - loss: 0.2648 - mae: 0.2648 - val_loss: 0.2565 - val_mae: 0.2565
Epoch 48/50
3/3 [=====] - 0s 14ms/step - loss: 0.2541 - mae: 0.2541 - val_loss: 0.6160 - val_mae: 0.6160
Epoch 49/50
3/3 [=====] - 0s 15ms/step - loss: 0.3663 - mae: 0.3663 - val_loss: 0.4216 - val_mae: 0.4216
Epoch 50/50
3/3 [=====] - 0s 13ms/step - loss: 0.3388 - mae: 0.3388 - val_loss: 0.6467 - val_mae: 0.6467
<keras.callbacks.History at 0x7fe9f02d7dd0>
```

Podemos observar que después de 50 épocas, donde en cada época se recorre todo el conjunto de entrenamiento, se logra tener un error de aproximadamente 0.3388 en el cálculo de la temperatura en Fahrenheit a partir de un valor en Celsius. Esto si bien puede mejorarse por distintos hyperparametros, para un primer ejercicio es más valioso entender la mecánica de la regresión lineal.

Realizamos predicciones con el modelo entrenado usando el conjunto de testeo de prueba.

Podemos observar que el promedio de los valores de predicción del modelo es muy aproximado al promedio de los valores de etiqueta del conjunto de testeo. El error

```
▶ modelo.predict(x_test).mean(),y_test.mean()
[] (192.0533, 192.7)
```

observado es muy bajo, por lo que la red neuronal podrá generalizar satisfactoriamente la relación entre Celsius y Fahrenheit.

Referencias

1. James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning* (Vol. 112, p. 18). New York: springer.
2. VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc."
3. Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media.
4. Certificación Universitaria en Data Science. Mundos E - Universidad Nacional de Cordoba.
5. Certificación en Machine Learning - Machine Learning for Beginners. Analytics Vidhya.

Made with ❤ Ignacio Bosch