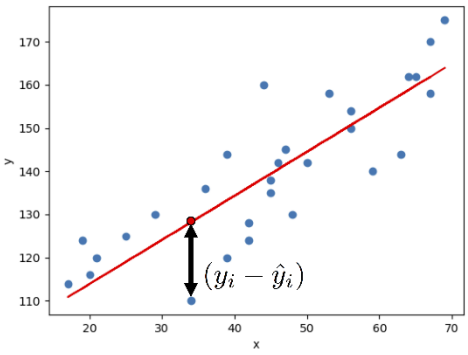
Centro Universitario de Ciencias Exactas e Ingenierías

Universidad de Guadalajara

Actividad 3: Regresiones

Aprendizaje Máquina



Alumno: Samuel David Pérez Brambila

Código: 222966286

Profesora: Karla Ávila Cárdenas

Sección: D01

Fecha de Entrega: 15 de Septiembre de 2024

Introducción

El aprendizaje máquina, machine learning o también llamado aprendizaje automático es “una disciplina del campo de la Inteligencia Artificial que, a través de algoritmos, dota a los ordenadores de la capacidad de identificar patrones en datos masivos y elaborar predicciones (análisis predictivo).” (Iberdrola, s.f.)

El aprendizaje máquina o automático, según Iberdrola (s.f.) se dividen en 3 principales categorías:

- **Aprendizaje supervisado:** Estos algoritmos cuentan con un aprendizaje previo basado en un sistema de etiquetas asociadas a datos que les permiten tomar decisiones o hacer predicciones, por ejemplo, un detector de spam que etiqueta un e-mail como spam o no dependiendo de los patrones que ha aprendido del histórico de correos.
- **Aprendizaje no supervisado:** Estos algoritmos no cuentan con un conocimiento previo, se enfrentan al caos de datos con el objetivo de encontrar patrones que permitan organizarlos de cierta manera. Por ejemplo, en marketing se utilizan para extraer patrones de datos masivos provenientes de las redes sociales y de esa forma crear campañas publicitarias altamente segmentadas.
- **Aprendizaje por refuerzo:** Su objetivo es que un algoritmo aprenda a partir de la propia experiencia. Esto se refiere a que sea capaz de tomar la mejor decisión ante diferentes situaciones de acuerdo con un proceso de prueba y error. En la actualidad se utiliza para posibilitar el reconocimiento facial, clasificar secuencias de ADN y para diagnósticos médicos.

En el presente trabajo nos enfocaremos en el aprendizaje supervisado, que de acuerdo con IBM (s.f.), se divide en 2 tipos de problemas: clasificación y regresión.

- La clasificación utiliza un algoritmo para asignar con precisión los datos de prueba en categorías específicas. Reconoce entidades específicas dentro del conjunto de datos e intenta sacar algunas conclusiones sobre cómo deben etiquetarse o definirse esas entidades.
- La regresión se utiliza para comprender la relación entre variables dependientes e independientes. Se utiliza comúnmente para hacer proyecciones, como los ingresos por ventas de un negocio determinado.

Habiendo ya diferenciado estas divisiones del aprendizaje supervisado, cabe destacar que nos adentraremos en esta ocasión en la regresión.

Pero así como pudimos clasificar el machine learning y luego el aprendizaje supervisado, la regresión también tiene diferentes tipos, donde según MailChimp (s.f.) los más habituales son:

- Regresión lineal sencilla: Este tipo de aprendizaje automático suele ser el primer tipo de regresión de aprendizaje automático que se aprende. Con un modelo de regresión lineal simple, existe una relación entre una única variable de entrada y una única variable de salida separada. El modelo de aprendizaje automático tratará de averiguar cómo están relacionadas ambas variables.
- Regresión lineal múltiple: Si se tienen más de dos variables, es entonces que nos enfrentamos a una regresión lineal múltiple. El objetivo del modelo de aprendizaje automático es intentar encontrar la relación entre varias variables de entrada y una variable de salida por el otro lado.
- Regresión polinomial: Si se tienen datos que no se pueden separar de una manera lineal, es posible que se tenga que utilizar la regresión polinomial. Este tipo de regresión es similar a la regresión lineal pero, en lugar de una línea, se intenta encontrar una curva que se ajuste a todos los puntos de datos.
- Regresión de vectores de soporte: La regresión de vectores de soporte es un algoritmo que permite predecir valores discretos teniendo en cuenta la información que se tiene en manos. Este tipo de regresión tiene como objetivo encontrar la línea que mejor se adapte al modelo. Esa línea cruzará el número máximo de puntos representados por el conjunto de datos.
- Regresión de árboles de decisión: El objetivo de la regresión de árboles de decisión es crear modelos de clasificación que adopten la estructura de un árbol. La regresión comenzará con el conjunto de datos y se dividirá en subconjuntos cada vez más pequeños. Esto expandirá poco a poco el árbol, creando nodos de decisión y nodos de hoja. El nodo de decisión tiene dos o más ramas fuera de él, mientras que el nodo de hoja representa una decisión única y selectiva.
- Regresión de bosque aleatorio: Si se tienen varios algoritmos de aprendizaje automático que se necesitan combinar en un solo modelo, se tiene una regresión de bosque aleatorio. La regresión de bosque aleatorio utiliza el aprendizaje supervisado para combinar predicciones de varios algoritmos en un solo algoritmo que se puede aplicar a varias situaciones. El algoritmo final adoptará la forma de un árbol, con diferentes funciones que se utilizan para dividir los nodos a medida que se desplace por el árbol. Luego se deberían incluir predicciones finales en la parte inferior.

En el siguiente apartado, nos enfocaremos en el desarrollo práctico y teórico de estos métodos de regresión, ilustrando cómo se implementan y ajustan utilizando herramientas modernas como SciKit-Learn. Analizaremos detalladamente las características, ventajas y limitaciones de cada tipo de regresión, con ejemplos que demuestran su capacidad para resolver problemas reales de predicción y modelado de datos en Machine Learning.

Contenido de la Actividad

Regresión

Los modelos de regresión lineal realizan predicciones numéricas sobre un objetivo basadas en un conjunto de datos clasificados según una o más características.

Modelos de regresión

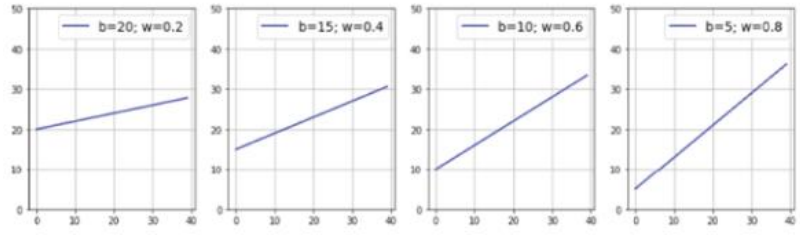
En esta sección se explica, mediante ejemplos, los modelos de regresión más comunes y útiles. Algunos se implementan desde cero para proporcionar:

- Los detalles de su implementación
- Los algoritmos de programación de machine learning
- Un entendimiento más profundo de las bibliotecas existentes
- La flexibilidad para presentar gráficamente los resultados
- El análisis de la evolución del modelo de aprendizaje

Regresión lineal (desde cero)

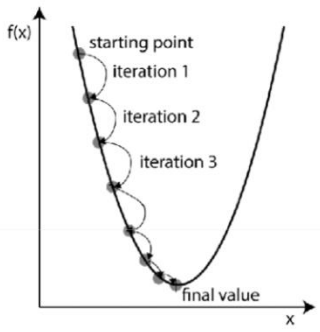
La regresión lineal desarrolla un modelo que ajusta las muestras usando una recta. Matemáticamente, la regresión lineal se representa mediante la ecuación y = wx + b. Los híper-parámetros aquí son (w,b). La pendiente está determinada por w, mientras que b establece el desplazamiento (offset): distancia vertical al eje x. El siguiente código muestra cuatro rectas diferentes con offset que va desde 20 a 5 (ver eje y) y pendientes crecientes desde 0.2 a 0.8

```
import matplotlib.pyplot as plt
import numpy as np
fig, axs = plt.subplots(1,4, figsize=(15,4))
X = np.array(range(40))
for fig,b,w in zip([0,1,2,3],[20,15,10,5],[0.2,0.4,0.6,0.8]):
y = b + w*X
axs[fig].plot(X, y, 'b-', label = 'b='+format(b)+'; w='+format(w))
axs[fig].set_ylim(bottom=0,top=50),axs[fig].legend();
axs[fig].legend(prop={'size':14}); axs[fig].grid();
plt.show()
```



La regresión lineal se puede llevar a cabo usando diferentes métodos (implementaciones). En esta ocasión se usará el conocido método del gradiente descendente. Este algoritmo busca un mínimo en el híper-espacio de errores; esto es: busca los errores más bajos que le sea posible encontrar. Se habrá encontrado

una solución adecuada si se alcanza un mínimo global en un espacio cóncavo de errores o un mínimo local razonable en un espacio convexo de errores. La siguiente figura muestra el concepto.



A continuación, se muestran las ecuaciones básicas que dan soporte al método de regresión lineal de gradiente descendente. La ecuación 1 muestra la función de coste, o función de pérdida (loss) que deseamos minimizar: el error existente en la predicción se calcula restando el valor de la muestra y_i menos la predicción del modelo wx_i+b. En vez de usar el valor absoluto, se utiliza el cuadrado para: a) que sea posible derivar la ecuación, y b) penalizar errores grandes. Las ecuaciones 2 y 3 muestran, respectivamente, las derivadas parciales de la función de coste respecto de b y w. Ambas derivadas muestran la manera de alcanzar el mínimo de la función de error en su espacio tridimensional.

$$loss(W,b) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - b - wx_i\right)^2 \quad (1)$$

$$\frac{\partial loss(w,b)}{\partial b} = -2\frac{1}{N}\sum_{i=1}^{N}(y_i - b - wx_i) \quad (2)$$

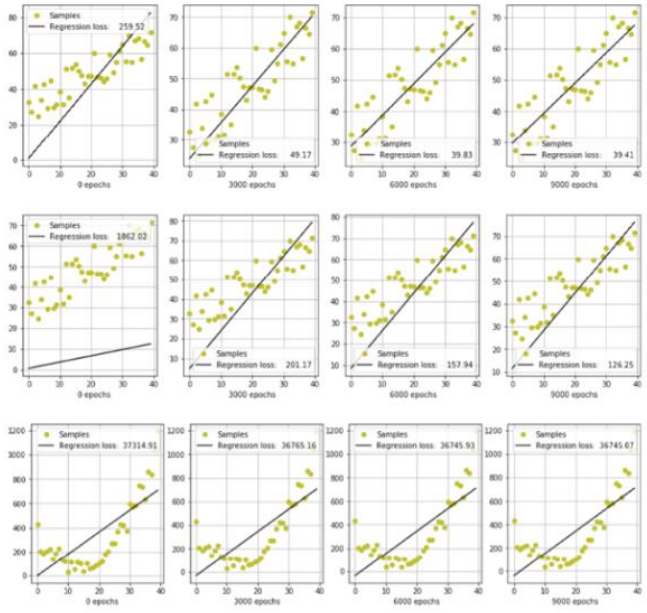
$$\frac{\partial loss(w,b)}{\partial w} = -2\frac{1}{N}\sum_{i=1}^{N}x_i(y_i - b - wx_i) \quad (3)$$

Las ecuaciones anteriores se implementan en el código mostrado a continuación, con *gradient_regression(X, y, alpha, b, w)*. Nótese el uso del parámetro *alpha*; es importante debido a que determina el tamaño de los pasos realizados para alcanzar el mínimo en la función de error, si *alpha* es muy pequeño, el algoritmo necesitará demasiado tiempo para alcanzar el mínimo; si *alpha* es demasiado grande, podría 'pasarse' el mínimo y oscilar a su alrededor. La función del modelo *model(X, y, alpha, b, w, epochs)* muestra el concepto de *"epoch"* (ciclo, época): cada "epoch" se corresponde con el proceso de todas las muestras de datos; el modelo progresa en cada epoch. En este código se han usado 9000 epochs con el objetivo de encontrar una solución adecuada. La función *prediction(x, b, w)* devuelve la predicción para *y* procesando el modelo *(b, w).* La última función en el código: *loss(X, y, b, w)* devuelve las diferencias cuadráticas medias (MSD) como una medida de calidad del modelo obtenido aplicado a las muestras correspondientes. El programa muestra cuatro

grupos de información. El primer grupo visualiza los resultados de la regresión lineal correspondientes al modelo existente al inicio (cero ciclos o epochs), el siguiente cuando se alcanza el proceso de 3000 ciclos (epochs) y posteriormente al alcanzar 6000 y 9000 ciclos. Se puede observar que el algoritmo iterativo mejora el modelo a medida que el número de ciclos crece. El segundo grupo de gráficos repite el mismo experimento, pero en este caso se ha elegido un valor del parámetro alpha excesivamente pequeño. Se puede observar que el algoritmo también itera hacia la solución, pero muy despacio debido al valor incorrecto de alpha. El tercer grupo de gráficos contiene el conjunto de muestras de datos que no pueden ser clasificadas correctamente mediante un enfoque de regresión lineal. Esto se puede comprobar visualmente y se puede también establecer a partir de los grandes valores de error que muestran los resultados de coste (loss). Finalmente, el cuarto grupo de información muestra los valores de predicción para las muestras de x con valores 10 y 20. Viendo el gráfico cuatro en el grupo uno (primera fila de gráficos), podemos observar que los valores de predicción son consistentes con el rango esperado.

```
import matplotlib.pyplot as plt
import random
import numpy as np
def gradient_regression(X, y, alpha, b, w):
dw = 0.0; db = 0.0
# We make the model by using all the samples
for i in range(len(X)):
aux = -2.0*(y[i]-(w*X[i]+b))
db = db + aux # this solver can easily overflow
dw = dw + X[i]*aux # this solver can easily overflow
aux = 1.0/float(len(X))
b = b - aux*db*alpha
w = w - aux*dw*alpha
return b,w
def gradient_regression2(X, y, alpha, b, w):
  aux = -2*(y-(w*X+b)).sum()
  b = b - alpha*aux/float(len(X))
  w = w - alpha*aux/float(len(X))
  return b,w
  def plot(fig, X, y, b, w, epochs):
  axs[fig].plot(X, y, 'yo', label = 'Samples')
  X = np.array(X)
  axs[fig].plot(X,w*X+b, 'k-', label = 'Regression loss:
  '+'{:9.2f}'.format(loss(X,y,b,w)))
  axs[fig].set_xlabel('{:5.0f}'.format(epochs) + ' epochs')
  axs[fig].legend(); axs[fig].grid();
  return
  def model(X, y, alpha, b, w, epochs):
  fig = 0
  for e in range(epochs):
  b, w = gradient_regression(X, y, alpha, b, w)
  if e % 3000 == 0:
  plot(fig, X, y, b, w, e)
  fig += 1
  return b, w;
  def prediction(x, b, w):
  return (x*w+b)
  def loss(X, y, b, w):
  sum = 0
  for i in range(len(X)):
  sum += (y[i]-prediction(X[i], b, w)) ** 2
  return sum/len(X)
  def create_samples(n):
  y = []; X = list(range(40))
  for i in range(len(X)):
  y.append(20+X[i]+random.random()*20)
  return X, y
  def create_samples2(n):
  X = np.array(list(range(40)))
  y = 20+X+np.random.rand(40)*20
  return X, y
  X, y = create_samples2(40)
  # in this example, we do not change epochs' value, since we just show
  #plots for the values (0, 3000, 6000, 9000)
```

```
EPOCHS = 9001;
# linear regression evolution
fig, axs = plt.subplots(1,4, figsize=(15,4))
# b and w parameters can be better initialized
b, w = model(X, y, 0.001, random.random(), random.random(), EPOCHS)
# linear regression using a wrong alpha value
fig, axs = plt.subplots(1,4, figsize=(15,4))
# b and w parameters can be better initialized
model(X, y, 0.0001, random.random(), random.random(), EPOCHS)
y = []
for i in range(len(X)):
y.append((X[i]-10)**2+random.random()*20*abs(len(X)/2-i))
# loss evolution
fig, axs = plt.subplots(1,4, figsize=(15,4))
# b and w parameters can be better initialized
model(X, y, 0.001, random.random(), random.random(), EPOCHS)
plt.show()
print(prediction(10, b, w))
print(prediction(20, b, w))
```




```
39.33286017829371
48.9970152111386
```

Regresión lineal usando SciKit

Para ilustrar este ejemplo de regresión lineal usando las librerías de SciKit, usaremos el dataset *Boston* y el modelo lineal *linear_model. LinearRegression*. Los principales parámetros de su constructor son:

- *Normalize*: para normalizar los datos de entrada. Su valor por defecto es True ya que la regresión lineal es muy sensible al hecho de que haya diferentes escalas en los datos de entrada.
- *Copy_X*: es True, X se copiará; si no, puede ser rescrito. Su valor por defecto es True.
- Atributos: los coeficientes w y b explicados en la sección anterior. SciKit los denomina coef_ (w) e intercept_ (b).

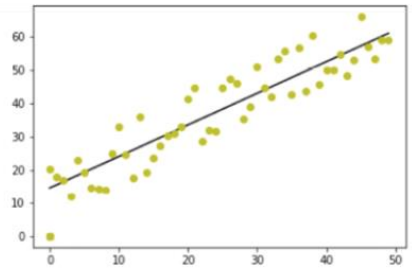
Los métodos que realizan el ajuste (fit) y la predicción (predict) en la regresión lineal de SciKit reciben matrices como argumentos. El siguiente código genera muestras lineales aleatorias y las empaqueta en matrices de dos dimensiones, preparadas para ser usadas por los métodos de ajuste y predicción. Al ejecutar el constructor

LinearRegression() se obtiene la instancia linear_regresssion. El modelo se crea con el uso del método fit(X,y). Una vez que el modelo ha sido creado, podemos acceder a los valores de la pendiente y del desplazamiento usando los atributos coef_ (w) e intercept_ (b). Dado que tanto los datos de entrada como los de salida son matrices de dos dimensiones, los atributos resultantes son codificados en forma vectorial y se puede acceder a los valores individuales a través de w[0][0] y b[0]. Los resultados muestran una pendiente de 0.94 y un punto de corte de 14.62. Por último, el método predict(X) devuelve el resultado esperado de la regresión, como podemos observar en la figura.

```
from sklearn import linear_model
import matplotlib.pyplot as plt
import numpy as np
n_samples = 50
X = np.zeros((n_samples,2))
y = np.zeros((n_samples,2))
X[:,0] = np.array(list(range(n_samples)))
y[:,0] = 5+X[:,0]+np.random.rand(n_samples)*20
# this is our chosen model
linear_regression = linear_model.LinearRegression()
linear_regression.fit(X, y) # train the model
w = linear_regression.coef_
b = linear_regression.intercept_
print(w[0][0])
```

```
print(b[0])
# y_predicted = []
# for i in range(len(X)):
# y_predicted.append(X[i,0]*w[0]+b)
y_predicted = linear_regression.predict(X)
plt.plot(X, y_predicted, 'k-')
plt.plot(X, y, 'yo')
plt.show()
```

```
0.9458308328057163
14.625240638309428
```



La ecuación de la diferencia cuadrática medida es:

$$MSD(X,y)=\frac{1}{n}\sum_{i=1}^{n}\left(y_i-prediction(x_i)\right)=\frac{1}{n}\sum_{i=1}^{n}\left(y_i-(wx_i-b)\right)^2$$

Regresión Polinómica (desde cero)

La regresión lineal nos permite crear un modelo lineal utilizando un polinomio de grado uno: $y = ax + b$. Algunos datos no pueden ser entrenados de esta manera y necesitan un tipo de regresión más complejo. La regresión polinómica puede ser definida como: $y=a_0+a_1x+a_2x^2+a_3x^3$... A continuación, vamos a implementar un algoritmo que utiliza regresión polinómica de grado 2: $y=a_0+a_1x+a_2x^2$. Usando el algoritmo del gradiente descendente, primero establecemos la función de coste (ecuación 1).

$$loss(a_0,a_1,a_2)=\sum_{i=1}^{N}\left(y_i-a_0-a_1x_i-a_2x_i^2\right)^2 \quad (1)$$

$$\frac{\partial loss(a_0,a_1,a_2)}{\partial a_0}=-2\sum_{i=1}^{N}(y_i-a_0-a_1x_i-a_2x_i^2) \quad (2)$$

$$\frac{\partial loss(a_0,a_1,a_2)}{\partial a_1}=-2\sum_{i=1}^{N}x_i(y_i-a_0-a_1x_i-a_2x_i^2) \quad (3)$$

$$\frac{\partial loss(a_0,a_1,a_2)}{\partial a_2}=-2\sum_{i=1}^{N}x_i^2(y_i-a_0-a_1x_i-a_2x_i^2) \quad (4)$$

$$\alpha_{11}a_0+\alpha_{12}a_1+\alpha_{13}a_2=\alpha_{14} \quad (5)$$

$$\alpha_{21}a_0+\alpha_{22}a_1+\alpha_{23}a_2=\alpha_{24} \quad (6)$$

$$\alpha_{31}a_0+\alpha_{32}a_1+\alpha_{33}a_2=\alpha_{34} \quad (7)$$

$$\alpha_{11}=N;\ \alpha_{12}=\sum x_i;\ \alpha_{13}=\sum x_i^2;\ \alpha_{14}=\sum y_i \quad (8)$$

$$\alpha_{21}=\sum x_i;\ \alpha_{22}=\sum x_i^2;\ \alpha_{23}=\sum x_i^3;\ \alpha_{24}=\sum x_iy_i \quad (9)$$

$$\alpha_{31}=\sum x_i^2;\ \alpha_{32}=\sum x_i^3;\ \alpha_{33}=\sum x_i^4;\ \alpha_{34}=\sum x_i^2y_i \quad (10)$$

$$\beta_{11}a_0+\beta_{12}a_1=\beta_{13} \quad (11)$$

$$\beta_{21}a_0+\beta_{22}a_1=\beta_{23} \quad (12)$$

$$a_0=\frac{\left(\beta_{22}/\beta_{12}\right)\beta_{13}-\beta_{23}}{\left(\beta_{22}/\beta_{12}\right)\beta_{11}-\beta_{21}} \quad (13)$$

$$a_1=\left(\beta_{23}-\beta_{21}a_0\right)/\beta_{22} \quad (14)$$

$$a_2=\left(\alpha_{24}-\alpha_{21}a_0-\alpha_{22}a_1\right)/\alpha_{23} \quad (15)$$

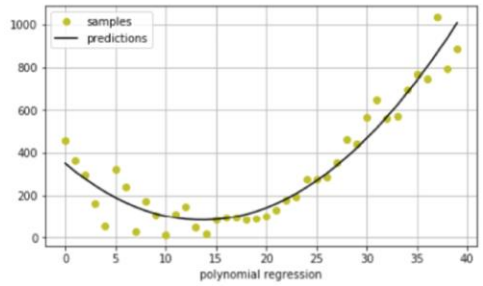
Posteriormente, con el objetivo de minimizar el error y poder encontrar la solución, derivamos la función de coste para cada uno de los híper- parámetros de regresión polinómica; en nuestro caso: a_0, a_1 y a_2. Los resultados de las derivaciones se muestran en las ecuaciones 2 a 4. A partir de aquí se define un sistema de tres ecuaciones (ecuaciones 5 a 10). La solución analítica del sistema nos lleva a los resultados de a_0, a_1 y a_2 mostrados en las ecuaciones 13 a 15.

Las ecuaciones obtenidas se implementan directamente en la función *polynomial_regression(X,y)* del siguiente código. Las predicciones se obtienen ejecutando el método *prediction(a0,a1,a2,x).* Las muestras se generan usando la

ecuación no lineal *(X[i]-10)*^{2} y utilizando una función aleatoria para evitar una función cuadrática perfecta. Los resultados visualizan la curva que ajusta y generaliza a los datos de entrada, representados mediante círculos.

```
import matplotlib.pyplot as plt
import random
# return the prediction of x data, by using kernel function
def polynomial_regression (X, y):
a12=0;a13=0;a14=0;a23=0;a24=0;a33=0;a34=0
for i in range(len(X)):
x_e2 = X[i]*X[i]; x_e3 = x_e2*X[i]; x_e4 = x_e3*X[i]
```

```
a12 += X[i]; a13 += x_e2; a14 += y[i]
a23 += x_e3; a24 += X[i]*y[i]
a33 += x_e4; a34 += x_e2*y[i]
a11 = len(X); a21 = a12; a22 = a13; a31 = a13; a32 = a23;
aux1 = a23/a13; aux2 = a33/a13;
b11 = aux1*a11-a21; b12 = aux1*a12-a22; b13 = aux1*a14-a24
b21 = aux2*a11-a31; b22 = aux2*a12-a32; b23 = aux2*a14-a34
a0 = ((b22/b12)*b13-b23)/((b22/b12)*b11-b21)
a1 = (b23-b21*a0)/b22
a2 = (a24-a21*a0-a22*a1)/a23
return a0, a1, a2
def prediction(a0, a1, a2, x):
return a0+a1*x+a2*x*x
# linear regression evolution
fig, axs = plt.subplots(figsize=(7,4))
# Create the samples
X = list(range(40))
y = []
for i in range(len(X)):
y.append((X[i]-10)**2+random.random()*20*abs(len(X)/2-i))
y_prediction = []
a0, a1, a2 = polynomial_regression(X,y)
for i in range(len(X)):
y_prediction.append(prediction(a0, a1, a2, X[i]))
axs.plot(X, y, 'yo', label='samples') # it plots samples
axs.plot(X, y_prediction,'k-', label='predictions') # it plots
predictions
axs.set_xlabel('polynomial regression')
axs.legend(); axs.grid();
plt.show()
```



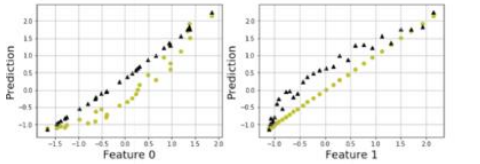
Regresión polinomial desde cero (enfoque de gradiente descendente)

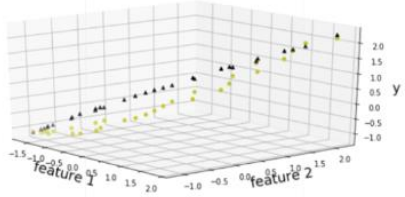
Hay un enfoque en machine learning para resolver la regresión polinómica que es más simple y general que la forma cerrada (closed form). En vez de ajustar a una curva polinómica, el enfoque usa una regresión lineal simple. El truco está basado en la inserción de algunas características nuevas que se pueden ajustar a la naturaleza no lineal de la distribución de las muestras.

En el ejemplo resultaría suficiente insertar una característica generada X^2, que dejaría la tarea de regresión lineal con las características de la X original y la nueva X^2. Naturalmente, ahora usamos la regresión lineal $y=w_0+w_1x_1+w_2x_2$ en vez de la

establecida previamente: $y=w_0+w_1x_1$. De este modo asignaremos un peso *w* a cada una de las características. Esta manera de operar puede ser generalizada a polinomios de mayor grado, añadiendo nuevas características no lineales. De hecho, la regresión con polinomios de alto grado funciona muy bien al añadir, no solamente las nuevas características básicas: X^2, X^3, X^4, etc. sino al insertar combinaciones de ellas: a^2b, b^2a, etc. El código mostrado más abajo contiene una implementación de la regresión por gradiente (Gradient Regression) para dos dimensiones. Ha sido codificado de dos maneras: *gradient_regression* y, en su forma condensada, usando operaciones de vectores y matrices: *gradient_regression2*, ambas implementaciones producen el mismo resultado. Después preparamos los datos lineales en *X[0]* y los datos cuadráticos para *y*. Para crear un modelo que prediga correctamente el objetivo no lineal *y*, añadimos una nueva dimensión de datos *X[1]* con forma cuadrática. Los resultados se presentan en las tres figuras siguientes, donde dos proyecciones en 2D y un gráfico en 3D muestran que el "truco" de regresión polinómica funciona correctamente para ajustar la función objetivo no lineal.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random
import numpy as np
def gradient_regression(X, y, alpha, b, w):
dw1 = 0.0; dw2 = 0.0; db = 0.0
for i in range(len(X[0])):
aux = -2.0*(y[i]-(w[0]*X[0][i]+w[1]*X[1][i]+b))
db = db + aux # this solver can easily overflow
dw1 = dw1 + X[0][i]*aux # this solver can easily overflow
dw2 = dw2 + X[1][i]*aux # this solver can easily overflow
aux = 1.0/float(len(X[0]))
b = b - aux*db*alpha
w[0] = w[0] - aux*dw1*alpha
w[1] = w[1] - aux*dw2*alpha
return b,w
def gradient_regression2(X, y, alpha, b, w):
aux = -2*(y-(w*X+b)).sum()
b = b - alpha*aux/float(len(X[0]))
w = w - alpha*aux/float(len(X[0]))
return b,w
def normalize(x):
# it can also be done by using the scypy (stats) zscore
mean = np.mean(x)
sdeviation = np.std(x)
y = (x-mean)/sdeviation
return y
np.random.seed(10)
# Create the samples
X = [[],[]]
y = []
result = []
for i in range(30):
X[0].append(i+random.random()*5)
X[1].append(i**2+random.random()*5)
#y.append((X[0][i]+20 + random.random()*10)+(X[1][i]-10 +
random.random()*10))
y.append((i**2+random.random()+50))
X[0] = normalize(X[0])
X[1] = normalize(X[1])
y = normalize(y)
alpha = 0.1
b = random.random()
w = np.random.rand(2,1);
w = np.array(w)
epochs = 4
for e in range(epochs):
b, w = gradient_regression(X, y, alpha, b, w)
result = w.T@X+b
fig, axs = plt.subplots(1,2, figsize=(12,4))
for i in [0,1]:
axs[i].scatter(X[i], y, c='y', marker='o')
axs[i].scatter(X[i], result, c='k', marker='^')
axs[i].set_xlabel('Feature ' + str(i), fontsize=18)
axs[i].set_ylabel('Prediction', fontsize=18)
axs[i].grid();
fig = plt.figure(figsize=(8, 4))
axs = Axes3D(fig, elev=20, azim=-40)
axs.scatter(X[0], X[1], y, c='y', marker='o')
axs.scatter(X[0], X[1], result, c='k', marker='^')
axs.set_xlabel('feature 1', fontsize=18)
axs.set_ylabel('feature 2', fontsize=18)
axs.set_zlabel('y', fontsize=18)
axs.grid();
plt.show()
```





Regresión de los K vecinos más cercanos (K-Nearest Neighbors o KNN) desde cero

KNN es un algoritmo que puede ser usado para abordar diferentes enfoques de machine learning, tales como la regresión o la clasificación. La operativa de la regresión es simple: hacemos predicciones basadas en las muestras más cercanas a la que pretendemos predecir; p. ej. si tenemos una lista (x, y) de muestras ordenadas (1.5, 6), (2.3, 4.5), (3, 8), (4.2, 9), (5.7, 10), y deseamos predecir la muestra x = 3.1, el algoritmo KNN realizará lo siguiente:

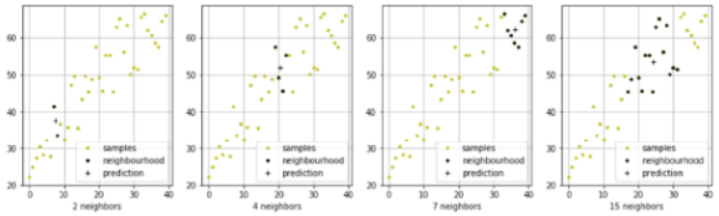
- Si k=1, entonces su único vecino es (3, 8), y la predicción es y = 8.
- Si k=2, los dos vecinos son: (3,8), (2.3, 4.5) y la predicción es y = (4.5+8)/2 = 6.25.
- Si k=3, los tres vecinos son: (3,8), (2.3, 4.5), (4.2, 9) y la predicción es y = (4.5+8+9)/3 = 7.16.
- Etc.

Este concepto se muestra gráficamente a partir de los resultados del siguiente código. El primer gráfico a la izquierda muestra la predicción para la muestra x = 7.6 usando 2 vecinos. El valor predicho aparece representado con una cruz, mientras que sus 2 vecinos son las muestras de color negro. El siguiente gráfico, a la derecha, muestra el valor predicho para x = 20.5 usando 4 vecinos, y así sucesivamente. La principal ventaja de KNN es su simplicidad. Sus principales desventajas son: las predicciones resultan lentas y la precisión de sus resultados es moderada.

El código que se presenta a continuación desarrolla el algoritmo KNN en su función *KNN_prediction (k, X, y, x, figure)*. Esta función admite los parámetros: *k*: número de vecinos, *X*: vector que contiene las coordenadas 'x' de las muestras, *y*: vector que contiene las coordenadas 'y' de las muestras, *x*: valor de la muestra 'x' para predecir su resultado 'y', figure: destinado simplemente a la representación gráfica. Las distancias están medidas usando la métrica euclídea: $(x - X[i])^2$; por eso debemos ordenar las muestras atendiendo a las distancias procesadas: *sort(distance)*. Por último, tomamos los primeros k elementos de *sorted_list* (que se corresponden a las *k* distancias más pequeñas). Al final del código hay cuatro llamadas a la función *KNN_prediction* y, cada resultado se dibuja en un gráfico de forma ordenada.

```
import matplotlib.pyplot as plt
import random
# Create the samples
X = list(range(40))
y = []
for i in range(len(X)):
```

```
y.append(20+X[i]+random.random()*20)
# linear regression evolution
fig, axs = plt.subplots(1,4, figsize=(15,4))
# return the prediction of x data, by using k neighbours
def KNN_prediction(k, X, y, x, figure):
distance = []
for i in range(len(X)):
distance.append((X[i],y[i],(x-X[i])**2)) # create pairs (X[i],
y[i], distance from x to X[i])
# sorts distance vector attending to the set of distances from x to X[i]
sorted_list = sort(distance)
x_neighbourhood, y_neighbourhood = [], [];
y_prediction = 0.0
for i in range(k): # k-neighbours
y_prediction += sorted_list[i][1]
x_neighbourhood.append(sorted_list[i][0])
y_neighbourhood.append(sorted_list[i][1])
y_prediction /= k
plot(figure,X,y,'y.', "samples") # plot samples
# plot the k neighbours
plot(figure,x_neighbourhood,y_neighbourhood,'k.', "neighbourhood")
# plot the (x,y_prediction) prediction
plot(figure,x,y_prediction,"k+", 'prediction')
axs[figure].set_xlabel('{:1.0f}'.format(k) + ' neighbors')
return y_prediction
def sort(unsorted_list):
return (sorted(unsorted_list, key = lambda x: x[2]))
def plot(fig, X, y, parameters, label):
axs[fig].plot(X, y, parameters, label = label)
axs[fig].legend(); axs[fig].grid();
return
KNN_prediction(2,X, y, 7.6, 0)
KNN_prediction(4,X, y, 20.5, 1)
KNN_prediction(7,X, y, 36.2, 2)
KNN_prediction(15,X, y, 24.3, 3)
plt.show()
```

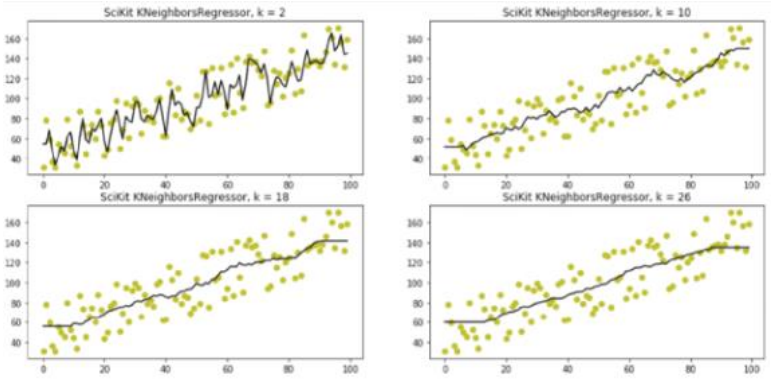


Regresión por K vecinos más cercanos (KNN) usando librerías SciKit

La regresión KNN en SciKit está implementada en el paquete *neighbors* del módulo *KNeighborsRegressor*. El siguiente código genera los datos de entrada y el objetivo de salida; nótese la inclusión de una nueva dimensión en los datos *X*: es necesario usar una matriz como argumento para el método de ajuste (*fit*). Creamos una instancia del regresor *knn_regressor*, aportando el tamaño de la vecindad seleccionada: *n_neighbors*. Después ajustamos nuestro modelo usando la instancia *knn_regressor*, los datos *X* y el objetivo *y*. Usando el modelo es posible obtener la predicción para diferentes muestras: *predict(X)*.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors
import random
X = np.array(list(range(100)))[:, np.newaxis]
y = []
for i in range(len(X)):
```

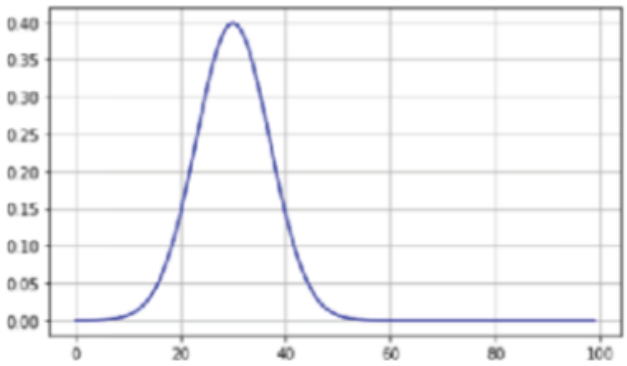
```
y.append(20+X[i]+random.random()*60)
fig, axs = plt.subplots(2,2, figsize=(15,7))
for i,n_neighbors in zip([0,1,2,3], [2,10,18,26]):
knn_regressor = neighbors.KNeighborsRegressor(n_neighbors)
knn_model = knn_regressor.fit(X, y)
y_prediction = knn_model.predict(X)
axs[int(i/2),i%2].scatter(X, y, c='y')
axs[int(i/2),i%2].plot(X, y_prediction, c='k')
axs[int(i/2),i%2].set_title('SciKit KNeighborsRegressor, k = '+
str(n_neighbors) )
plt.show()
```



Regresión Kernel Gaussiana (Gaussian Kernel Regression) desde cero

La regresión kernel estima la expectativa condicionada de una variable aleatoria. Permite encontrar relaciones no lineales entre variables aleatorias X e Y. La técnica de regresión kernel gaussiana es una regresión no lineal. Este tipo de regresión no requiere de ningún aprendizaje iterativo: usa el concepto KNN de utilización de los vecinos cercanos a la muestra predicha. Hay dos diferencias principales entre regresión kernel y los k vecinos más cercanos: 1) La regresión kernel toma una media ponderada de las muestras cercanas, y 2) La regresión kernel usa todas las muestras como vecinos.

Para ponderar las muestras cercanas usamos la conocida distribución gaussiana, y esta es la razón de que se denomine Regresión Kernel Gaussiana. En la siguiente figura se representa una función gaussiana; si deseamos realizar una predicción para la muestra x=30 se puede ver que las muestras de los vecinos más cercanos tomarán valores de pesos (ponderación) grandes, mientras que los que estén a mayor distancia tomarán valores de pesos pequeños.



La Regresión Kernel Gaussiana puede ser modelada usando las ecuaciones 2 a 5 mostradas a continuación. La función gaussiana se define en la ecuación 3, donde z_i contiene la media y la varianza seleccionadas (ecuación 1). La regresión kernel gaussiana usa el valor x de la muestra de predicción como la media de la función gaussiana, ya que lo que queremos es centrar la función en esta muestra (x=30 en el gráfico anterior). El algoritmo de kernel gaussiano hace uso del parámetro b para

establecer la varianza de la función gaussiana y su distribución resultante de los pesos (ecuación 2). Cada peso se obtiene aplicando la ecuación 4, donde el peso asignado a la predicción i (numerador) es ponderado con la media de los pesos de todas las muestras (denominador). Por último, la predicción (ecuación 5) se obtiene como la media ponderada de los objetivos (en este ejemplo, la vecindad es el conjunto de todas las muestras).

$$z_i = \frac{(x_i - \mu)}{\sigma} \quad (1)$$

$$z_i = \frac{(x_i - x)}{b} \quad (2)$$

$$Gaussian(z_i) = \frac{1}{\sqrt{2\pi}} e^{(-z_i^2/2)} \quad (3)$$

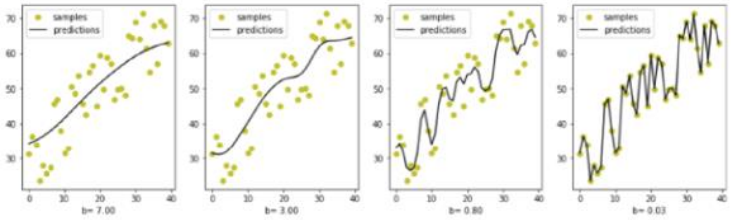
$$w_i = \frac{Gaussian(z_i)}{\frac{1}{N}\sum_{j=1}^{N} Gaussian(z_j)} \quad (4)$$

$$y = \frac{\sum_{i=1}^{N} (w_i y_i)}{\sum_{i=1}^{N} w_i} \quad (5)$$

El siguiente código desarrolla la regresión kernel gaussiana. Su función *kernel(z)* devuelve el valor gaussiano de z (ecuación 3). La función w(b, i, X, x) implementa la ecuación 4; usa la función *kernel*, aportando la muestra a predecir x como media gaussiana y el parámetro b como la varianza gaussiana (ecuación 2). Por último, la función *kernel_function(b, X, y, x, figure)* implementa la ecuación 5. Al final del código se prueba la función *kernel_function* usando diferentes valores de varianza: (7, 3, 0.8, 0.03). Los gráficos resultantes muestran una buena generalización cuando la varianza es 7 y 3. También muestran sobreajuste para los valores bajos de varianza.

```
import matplotlib.pyplot as plt
import random
import math
# Create the samples
X = list(range(40))
y = []
```

```
for i in range(len(X)):
y.append(20+X[i]+random.random()*20)
# linear regression evolution
fig, axs = plt.subplots(1,4, figsize=(15,4))
# return the prediction of x data, by using kernel function
def kernel_function(b, X, y, x, figure):
y_prediction = 0
for i in range(len(X)):
y_prediction += w(b, i, X, x)*y[i]
y_prediction /= len(X)
return y_prediction
def w(b, i, X, x):
denom = 0;
for j in range(len(X)):
denom += kernel((X[j]-x)/b)
return (len(X)*kernel((X[i]-x)/b))/denom
def kernel(z):
return math.exp(-(z*z)/2)/math.sqrt(2*math.pi)
def plot(fig, X, y, parameters, label):
axs[fig].plot(X, y, parameters, label = label)
axs[fig].legend(); axs[fig].grid();
return
for b, fig in zip([7, 3, 0.8, 0.03],range(4)):
y_prediction = []
for x in range(len(X)):
y_prediction.append(kernel_function(b, X, y, x, 0))
plot(fig, X, y, 'yo', "samples") # plot samples in figure
# plot the (x,y_prediction) prediction
plot(fig, X, y_prediction,"k-", 'predictions')
axs[fig].set_xlabel('b= ' + '{:4.2f}'.format(b))
plt.show()
```

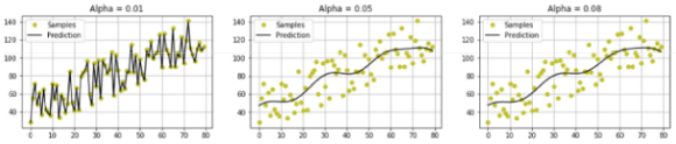


Regresión Kernel Gaussiana usando librerías SciKit

Para hacer una regresión kernel gaussiana usando la librería SciKit, seleccionamos el paquete *gaussian_process* e importamos el módulo *GaussianProcessRegressor*. Posteriormente elegimos alguno de los kernel existentes en el módulo *gaussian_process.kernel*. En este ejemplo usamos el kernel con la función de base radial (*radial basis function*: RBF). Como es habitual, primero creamos una instancia gp usando el constructor *GaussianProcessRegressor*. Después ajustamos los datos de entrada X y el objetivo y. Usando el modelo podemos predecir las muestras. El ejemplo visualiza los resultados del método para tres valores diferentes del híper-parámetro *alpha* del regresor. *Alpha* es la varianza de la función radial gaussiana.

```
import numpy as np
from matplotlib import pyplot as plt
import random
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
y = []
X = np.array(list(range(80)))[:, np.newaxis]
for i in range(len(X)):
y.append(20+X[i]+random.random()*50)
kernel = RBF(10, (0.01, 1e2))
fig, axs = plt.subplots(1,3, figsize=(17,3))
for i, alpha in zip([0,1,2], [0.01, 0.05, 0.08]):
```

```
gp = GaussianProcessRegressor(alpha=alpha, kernel = kernel)
gp.fit(X, y)
y_pred, sigma = gp.predict(X, return_std=True)
axs[i].plot(X, y, 'y.', markersize=11, label='Samples')
axs[i].plot(X, y_pred, 'k-', label='Prediction')
axs[i].legend()
axs[i].set_title('Alpha = ' + str(alpha))
axs[i].grid()
plt.show()
```



Regresión Ridge (forma cerrada)

La regresión Ridge, denominada también regresión de Tikhonov, es una regresión lineal regularizada usando L2. El término de regularización se usa para mantener los pesos del modelo tan pequeños como sea posible, al mismo tiempo que el algoritmo ajusta los datos. El término de regularización de la regresión Ridge tiene la forma:

$$L2 = \alpha \sum_{i=1}^{n} \theta_i^2$$

La función de coste (función de pérdida) de la regresión Ridge añade el término de regularización a la función de coste de la regresión lineal; en este sentido, los pesos del modelo se mantienen tan pequeños como sea posible. El híper-parámetro *alpha* determina el equilibrio entre la regularización y la relevancia de la regresión lineal.

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(y^i - \theta^T x^i \right)^2 + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

Al usar la regularización mantenemos los pesos pequeños y así el efecto de ruido y de valores aislados se minimiza. La regularización minimiza el sobreajuste y produce un modelo más generalizado.

La regresión Ridge puede ser resuelta en base al enfoque de gradiente descendente o usando su forma cerrada. En este caso usamos la forma cerrada:

$$\theta = (X^T X + \alpha I)^{-1} X^T y$$

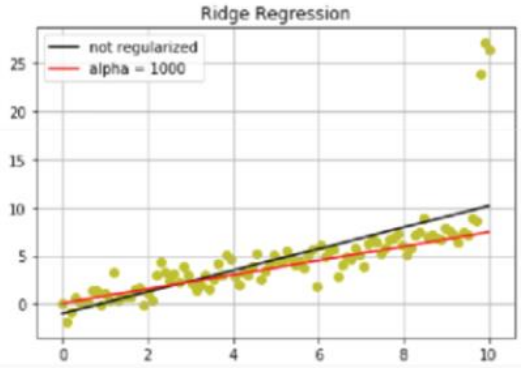
El siguiente código genera un conjunto de 100 muestras, incluyendo cuatro valores atípicos (outliers). Posteriormente se añade al vector X una columna rellenada con unos; se usa para mantener el primer coeficiente no regularizado. El bucle for ejecuta dos regresiones Ridge: la primera no regularizada (alpha=0) y la segunda

regularizada (L2=1000). Por último, las predicciones se hacen simplemente realizando el producto de los pesos y los datos de las muestras a predecir.

```
import numpy as np
```

```
import matplotlib.pyplot as plt
N = 100 # number of samples
X = np.linspace(0,10,N)
y = 0.8*X + np.random.randn(N)
for i in range(1,4):
y[-i]+=20-i # inserting outliners
#appending bias. The first coefficient must not been regularized
X = np.vstack([np.ones(N), X]).T
for alpha,color,label in zip([0,1000],['k','red'],['not
regularized','alpha = 1000']):
# Ridge regression
#w = np.linalg.pinv((np.dot(X.T, X)+alpha*np.eye(2)))@np.dot(X.T, y)
w = np.linalg.pinv((X.T@X+alpha*np.eye(2)))@X.T@y
y_result = X@w
plt.plot(X[:,1],y_result, color=color, label=label)
print(w)
plt.scatter(X[:,1], y, color='y')
plt.title('Ridge Regression')
plt.legend();plt.grid()
plt.show()
```

```
[-0.97506912  1.11685326]
[0.08335325 0.73846229]
```

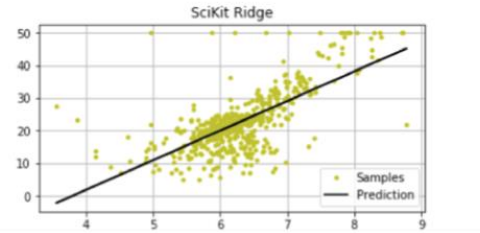


Ridge Regression usando librerías de SciKit

En el siguiente código aplicamos regresión Ridge al dataset *Boston*. Primero se carga el conjunto de datos y se selecciona su quinto atributo (número de habitaciones, "number of rooms"). Debido a que el método de ajuste necesita una matriz como argumento, añadimos una nueva dimensión a los datos (*X, [:, np.newaxis]*). Después se crea una nueva instancia ridge de linear_model.Ridge y se usa para configurar el modelo: *fit(X, y)*. Una vez que el modelo está creado, podemos acceder a los parámetros obtenidos: *coef_* e *intercept_*. Para llevar a cabo las predicciones podríamos procesar *y = X coef_ + intercept_*, pero en vez de esto vamos a usar el método de predicción proporcionado por SciKit.

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn import linear_model
import numpy as np
def plot(data2D, target1D, predict1D):
plt.figure(figsize=(6, 3))
plt.plot(data2D, target1D, 'y.', markersize=6, label='Samples')
plt.plot(data2D, predict1D, 'k-', label='Prediction')
plt.legend()
plt.title('SciKit Ridge')
plt.grid()
plt.show()
return
boston = load_boston()
X = boston.data[:,5] # The fifth Attribute is the number of romos
X = X[:, np.newaxis] # Regression fit needs a data matrix as argument
y = boston.target # Regression fit methods need a target vector
ridge = linear_model.Ridge(alpha=0.3)
ridge.fit(X, y)
print("Slope: "+str(ridge.coef_)+", intercept:"+str(ridge.intercept_))
y_pred = ridge.predict(X)
plot(X, y, y_pred)
```

```
Slope: [9.09116911], intercept:-34.60186769836396
```



Regresión Lasso usando librerías de SciKit

La regresión Lasso ("Least Absolute Shrinkage and Selection Operator") es similar a la regresión Ridge; simplemente cambia el término de regularización. En vez de la regularización L2, Lasso usa la regularización L1. La ecuación que aparece a continuación muestra la función de coste de Lasso. La regresión Lasso tiende a eliminar las características menos importantes, asignándoles valores cercanos a cero. A diferencia de la norma L2, la regularización L1 usada por Lasso no es derivable, pero el método del gradiente descendente funciona correctamente con ella.

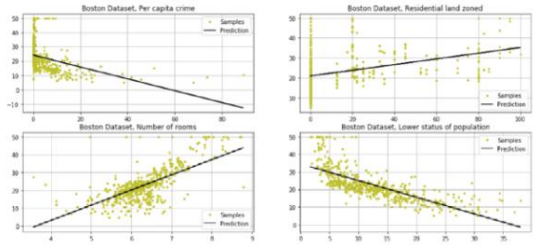
$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^i - \theta^T x^i\right)^2 + \alpha\sum_{i=1}^{n}\left|\theta_i\right|$$

El código que se muestra a continuación lleva a cabo la regresión Lasso en cuatro atributos diferentes en el conjunto de datos *Boston*. Como es habitual, importamos *load_boston* para insertarlo en la variable *Boston*: importamos también *linear_model* de Lasso. Después se crea una instancia del modelo usando un valor de *alpha* 0.3; en este caso, el valor del híper- parámetro no afecta mucho a la regresión debido a la distribución del conjunto de datos. Después de ajustar cada una de las cuatro regresiones, obtenemos las pendientes y los puntos de corte. Se puede ver que, como se esperaba, existe una correlación inversa entre el "crimen per cápita" y los precios de las propiedades; lo mismo que para capas sociales de menor status. Por

otra parte, el número de habitaciones presenta una alta correlación. Finalmente, el grado de propiedades residenciales parceladas presenta una pequeña correlación.

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.linear_model import Lasso
import numpy as np
fig, axs = plt.subplots(2,2, figsize=(16,7))
def plot(data2D, target1D, predict1D, i, labels):
axs[int(i/2),i%2].plot(data2D, target1D, 'y.', markersize=6,
label='Samples')
axs[int(i/2),i%2].plot(data2D, predict1D, 'k-', label='Prediction')
axs[int(i/2),i%2].legend()
axs[int(i/2),i%2].set_title('Boston Dataset, '+ labels[i])
axs[int(i/2),i%2].grid()
return
boston = load_boston()
y = boston.target # Regression fit methods need a target vector
CRIM = 0; ZN = 1; ROOMS = 5; LSTAT = 12
labels = ('Per capita crime','Residential land zoned','Number of
rooms','Lower status of population')
for i, attribute in zip(range(4),[CRIM, ZN, ROOMS, LSTAT]):
X = boston.data[:,attribute] # Representative attributes
# Regression fit methods need a data matrix as argument
X = X[:, np.newaxis]
lasso = Lasso(alpha=0.3)
lasso.fit(X, y)
print("Slope: "+str(lasso.coef_)+", intercept:
"+str(lasso.intercept_))
y_pred = lasso.predict(X)
plot(X, y, y_pred, i, labels)
```

```
Slope: [-0.41112746], intercept: 24.018425082751754
Slope: [0.14158737], intercept: 20.923858967771817
Slope: [8.49321328], intercept: -30.843933902927343
Slope: [-0.94415475], intercept: 34.47925604595717
```



Regresión Elastic Net usando librerías de SciKit

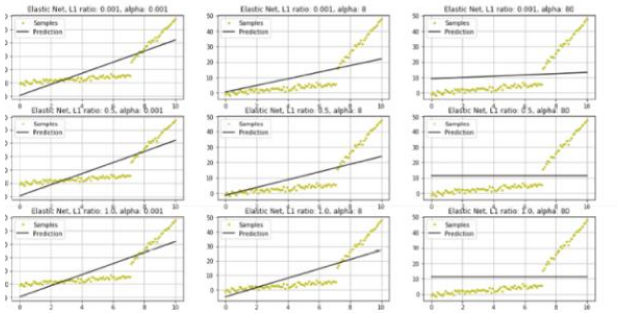
La regresión denominada *Elastic Net* combina la regresión *Ridge* y la regresión *Lasso*, ponderadas mediante el híper-parámetro *r*. El término de regularización de Elastic Net mezcla L2 (de la regresión Ridge) y L1 (de la regresión Lasso). La función de coste de Elastic Net es:

$$J(\theta)=\frac{1}{n}\sum_{i=1}^{n}\left(y^i-\theta^T x^i\right)^2+r\alpha\sum_{i=1}^{n}\left|\theta_i\right|+\frac{1-r}{2}\alpha\sum_{i=1}^{n}\theta_i^2$$

En el siguiente ejemplo se pone a prueba la regresión Elastic Net en un conjunto de datos generados que continen una gran cantidad de outliers.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import ElasticNet
fig, axs = plt.subplots(3,3, figsize=(18,9))
def plot(data2D, target1D, predict1D, row, column, title):
axs[row,column].plot(data2D, target1D, 'y.', markersize=6,
label='Samples')
```

```
axs[row,column].plot(data2D, predict1D, 'k-', label='Prediction')
axs[row,column].legend()
axs[row,column].set_title('Elastic Net, '+title)
axs[row,column].grid()
return
N = 100 # number of samples
X = np.linspace(0,10,N)
y = 0.8*X + np.random.randn(N)
for i in range(1,30):
y[-i]+=40-i # inserting outliners
X = np.vstack([np.zeros(N), X]).T
for l1_ratio, row in zip([0.001,0.5,1.0], [0,1,2]):
for alpha, column in zip([0.001,8,80], [0,1,2]):
elastic = ElasticNet(alpha=alpha, l1_ratio=l1_ratio)
elastic.fit(X, y)
y_pred = elastic.predict(X)
plot(X[:,1], y, y_pred, row, column, 'L1 ratio: '+str(l1_ratio)+
', alpha: '+str(alpha))
```



Análisis de calidad en la regresión lineal

Como recordatorio, indicamos que la medida de calidad *score* R^2 devuelve un valor entre 0 y 1, donde 1 significa una predicción perfecta y 0 se corresponde con el modelo de regresión más simple: predecir la media del conjunto de muestras de prueba. Asumiremos que el regresor probado no es peor que el regresor más simple. El *score* R^2 se denomina también *Coeficiente de Determinación*. La ecuación del R^2 *score* es:

$$R^2 = 1 - \frac{\sum_{i=0}^{n-1}(y_i - f(x_i))^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2}$$

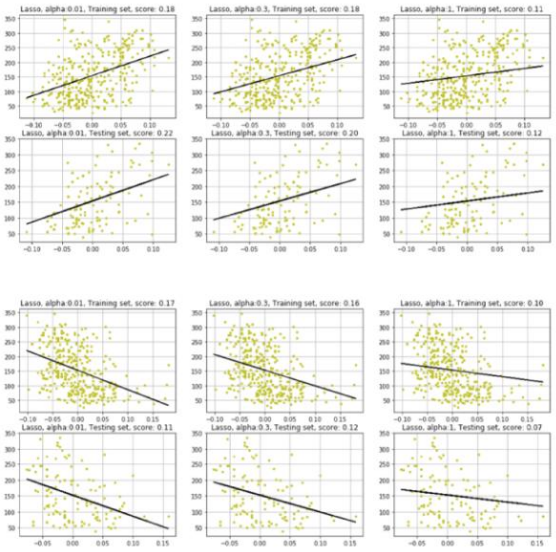
El siguiente código representa gráficamente varios resultados de la regresión *Lasso* e imprime el coeficiente de determinación de cada uno de ellos. Se ha usado el conjunto de datos de *Diabetes (Diabetes Dataset)*. El primer concepto importante es que tenemos que dividir las muestras existentes en los conjuntos disjuntos de entrenamiento (train) y de prueba (test). El método *train_test*, del paquete *model_selection* lleva a cabo esta tarea. Asignamos un valor a *random_state* para mantener los mismos conjuntos de entrenamiento y pruebas a través de diferentes ejecuciones. Después de obtener los conjuntos de entrenamiento y prueba, añadimos una nueva dimensión a los conjuntos de datos *X* (pero no a los objetivos), para asegurarnos de que los métodos de ajuste (*fit*) y predicción (*predict*) recibirán una matriz como argumento. Procederemos a probar de manera separada las

características *blood_pressure* y S3 del conjunto de datos de Diabetes. La función *model(X_train, y_train, X_test, y_test)* muestra los resultados para los conjuntos de entrenamiento y prueba usando tres valores diferentes de *alpha*. En el código interno de los bucles se crea una instancia *lasso* del regresor *Lasso*, para ajustar cada par de conjuntos de entrenamiento y prueba. El método *predict* devuelve los valores que predice el modelo, mientras que el método *score* muestra los valores de la puntuación (score) de calidad.

En las figuras mostradas podemos ver los gráficos correspondientes a *blood_pressure* (primer grupo de cuatro gráficos) y los gráficos correspondientes a la característica *s3* (segundo grupo de cuatro gráficos). En todos los resultados podemos observar que no hay sobreajuste, ya que existen muchas muestras y el modelo es muy simple (lineal: solamente dos coeficientes). Se puede observar, en los tres gráficos de entrenamiento de la presión arterial (los de más arriba) que los valores altos de *alpha* producen una degradación de calidad (valores más bajos en la puntuación): no es adecuado regularizar en exceso este modelo lineal, aplicado al conjunto de datos de *Diabetes*. En la siguiente fila que contiene tres gráficos de presión arterial (siguiente fila de gráficos), observamos que el modelo entrenado es adecuado para el conjunto de prueba: la distribución de las muestras de entrenamiento y prueba son similares. También se mejora la exactitud (*accuracy*) y los resultados de la puntuación (*score*) suben. El segundo grupo de gráficos, correspondientes a la característica *s3*, muestran una tendencia similar en la calidad relacionada con los valores crecientes de *alpha*. Por otra parte, las puntuaciones de prueba son menores que las de entrenamiento, y esto significan que el modelo entrenado no es adecuado para los datos existentes. Lo que muestran numéricamente los bajos valores de los resultados de *score* es lo que podemos observar en todos los gráficos: la solución lineal no es la adecuada para modelizar estas características del dataset *Diabetes*.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn import linear_model
def model(X_train, y_train, X_test, y_test):
fig, axs = plt.subplots(2,3, figsize=(16,7))
TRAINING = 0; TESTING = 1;
for set_type, label in zip((TRAINING, TESTING),
('Training set', 'Testing set')):
for alpha, column in zip([0.01,0.3,1], [0,1,2]):
lasso = linear_model.Lasso(alpha=alpha)
lasso.fit(X_train, y_train)
if set_type == TRAINING:
```

```
# Training accuracy
training_score = lasso.score(X_train, y_train)
y_pred = lasso.predict(X_train)
plot(axs, X_train, y_train, y_pred, alpha, set_type,
column, label+', score: {:3.2f}'.format(training_score))
else:
# Testing accuracy
testing_score = lasso.score(X_test, y_test)
y_pred = lasso.predict(X_test)
plot(axs, X_test, y_test, y_pred, alpha, set_type, column,
label+', score: {:3.2f}'.format(testing_score))
plt.show()
return
def plot(axs, data2D, target1D, predict1D, alpha,
set_type, column, label):
axs[set_type,column].plot(data2D, target1D, 'y.', markersize=6)
axs[set_type,column].plot(data2D, predict1D, 'k-')
axs[set_type,column].set_title('Lasso, alpha:' + str(alpha)+', '
+ label)
axs[set_type,column].grid()
return
diabetes = load_diabetes()
X = diabetes.data
y = diabetes.target
BLOOD_PRESSURE = 3; S3 = 6;
for feature in (BLOOD_PRESSURE, S3):
X_train, X_test, y_train, y_test = train_test_split(X[:,feature], y,
random_state = 60)
# Regression fit methods need a data matrix as argument
X_train = X_train[:, np.newaxis]
# The Score method needs a data matrix as argument
X_test = X_test[:, np.newaxis]
model(X_train, y_train, X_test, y_test)
print()
```



En este ejemplo final probamos varios regresores sobre el conjunto de datos de *Diabetes*. Haremos uso de la característica de presión arterial (*blood pressure*). La primera parte del código lleva a cabo todo lo necesario para importar el conjunto de datos, acondicionar sus valores y preparar los regresores. Después se carga el dataset, se obtienen los conjuntos de datos de entrenamiento y de prueba y se redimensionan para poder ser usados como argumentos en los métodos *fit* y *predict.* Se utiliza un vector de regresores para mantener las cinco instancias de regresores diferentes*: Lasso, Ridge, Gaussiano, KNN*, y lineal. Por último, se ejecuta un bucle que itera sobre los cinco regresores para ajustar cada modelo, predecir los resultados de entrenamiento y de prueba y procesar las medidas de calidad. Se han

elegido las medidas de calidad: *R2 score, Error Absoluto Medio (Mean Absolute Error) y Diferencias Cuadráticas Medias (Mean Squared Differences)*. Las medidas de calidad se obtienen comparando las predicciones objetivo (*train_predict* y *test_predict*) con sus correspondientes valores reales (*y_train* e *y_test*). Usamos algunos métodos de *sklearn.metrics: r2_score*, *mean_absolute_error,* y *mean_squared_error.*

```
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn import metrics
import numpy as np
from sklearn import linear_model, neighbors
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
diabetes = load_diabetes(); BLOOD_PRESSURE = 3;
X = diabetes.data
y = diabetes.target
X_train, X_test, y_train, y_test = train_test_split(X[:,BLOOD_PRESSURE],
y, random_state = 60)
X_train = X_train[:, np.newaxis]
X_test = X_test[:, np.newaxis]
regressors = []; labels = ['Lasso', 'Ridge', 'Gaussian',
'Neighbors','Linear']
regressors.append(linear_model.Lasso(alpha=0.1))
regressors.append(linear_model.Ridge(alpha=0.01))
regressors.append(GaussianProcessRegressor(alpha=1, kernel =
RBF(10, (0.01, 1e2))))
regressors.append(neighbors.KNeighborsRegressor(n_neighbors = 30))
regressors.append(linear_model.LinearRegression())
for regressor, label in zip(regressors,labels):
regressor.fit(X_train, y_train)
train_predict = regressor.predict(X_train)
test_predict = regressor.predict(X_test)
```

```
training_score = metrics.r2_score(y_train, train_predict)
testing_score = metrics.r2_score(y_test, test_predict)
training_MAE = metrics.mean_absolute_error(y_train,train_predict)
testing_MAE = metrics.mean_absolute_error(y_test, test_predict)
training_MSD = metrics.mean_squared_error(y_train,train_predict)
testing_MSD = metrics.mean_squared_error(y_test, test_predict)
print(label + ', R2 Score Training: {:.2f}'.format(training_score)+
', Testing: {:.2f}'.format(testing_score))
print(label + ', MAE. Training: {:.2f}'.format(training_MAE)+
', Testing: {:.2f}'.format(testing_MAE))
print(label + ', MSD. Training: {:.2f}'.format(training_MSD)+
', Testing: {:.2f}'.format(testing_MSD))
print()
```

```
Lasso, R2 Score Training: 0.18, Testing: 0.22
Lasso, MAE. Training: 57.67, Testing: 57.69
Lasso, MSD. Training: 4814.13, Testing: 4723.73
Ridge, R2 Score Training: 0.18, Testing: 0.22
Ridge, MAE. Training: 57.52, Testing: 57.31
Ridge, MSD. Training: 4810.09, Testing: 4696.12
Gaussian, R2 Score Training: 0.19, Testing: 0.22
Gaussian, MAE. Training: 56.94, Testing: 57.33
Gaussian, MSD. Training: 4750.07, Testing: 4683.97
Neighbors, R2 Score Training: 0.19, Testing: 0.21
Neighbors, MAE. Training: 57.20, Testing: 57.66
Neighbors, MSD. Training: 4790.83, Testing: 4757.69
Linear, R2 Score Training: 0.18, Testing: 0.22
Linear, MAE. Training: 57.49, Testing: 57.23
Linear, MSD. Training: 4809.92, Testing: 4689.85
```

Conclusiones

En conclusión, los diversos tipos de regresión ofrecen un amplio conjunto de herramientas para el análisis y predicción. Teniendo desde la simplicidad de la regresión lineal hasta la complejidad de los modelos de bosque aleatorio y Kernel Gaussiana. Cada método presenta ventajas específicas y aplicaciones concretas que los hacen indispensables en la resolución de diferentes problemas.

Herramientas como SciKit han democratizado el uso de estos modelos, permitiendo a los profesionales aplicar algoritmos de regresión de una manera eficiente y efectiva. No obstante, la elección del modelo adecuado depende en gran medida del contexto y naturaleza de los datos, y es de suma importancia entender las limitaciones inherentes a cada técnica. A medida que los datos siguen creciendo en volumen y complejidad, el aprendizaje continuo y la adaptación a nuevas metodologías seguirán siendo fundamentales para aprovechar al máximo el potencial de la regresión en machine learning.

Por todo lo anterior, el presente trabajo no solo subraya la importancia de los modelos de regresión, sino que también invita a seguir explorando sus capacidades y a innovar en su aplicación para resolver problemas actuales y futuros. La regresión, en sus múltiples formas, no solo es una herramienta analítica poderosa, sino también un puente hacia nuevas ideas y soluciones en el campo de la inteligencia artificial y más allá.

Referencias

- Bobadilla, J. (2020). *Machine Learning y Deep Learning Usando Python, Scikit y Keras* (1ª ed.). Ra-Ma Editorial; Ediciones de la U. https://elibro-net.wdg.biblio.udg.mx:8443/es/lc/udg/titulos/222698
- Iberdrola (s.f.). *Descubre los principales beneficios del 'Machine Learning'*. Iberdrola. Recuperado el 10 de Septiembre de 2024 de: https://www.iberdrola.com/innovacion/machine-learning-aprendizaje-automatico
- IBM (s.f.). *¿Qué es el aprendizaje supervisado?*. IBM. Recuperado el 10 de Septiembre de 2024 de: https://www.ibm.com/mx-es/topics/supervised-learning
- MailChimp (s.f.). *Comprender la regresión del aprendizaje automático: Una guía completa*. Intuit MailChimp. Recuperado el 10 de Septiembre de 2024 de: https://mailchimp.com/es/resources/machine-learning-regression/