Tecnológico de Monterrey

Inteligencia artificial avanzada para la ciencia de datos I (Gpo 101)

Momento de Retroalimentación: Módulo 2 Implementación de una técnica de aprendizaje máquina sin el uso de un framework. (Portafolio Implementación)

Maxime Vilcocq Parra

A01710550

Implementación de una técnica de aprendizaje máquina sin el uso de un framework:

Para esta entrega estaré programando el algoritmo de gradient descent para generar un modelo predictivo (calculando coeficientes de distintas variables) reduciendo el MSE (mean squared error).

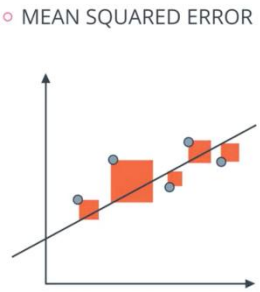
Para mi programa estaré utilizando el dataset de abalone, el cual es un set de datos para predecir la edad de este tipo de árbol a partir de medidas físicas (en lugar del tedioso método tradicional). Sin embargo, el código debe de funcionar con otros sets de datos siempre y cuando se introduzcan de forma correcta.

Gradient descent:

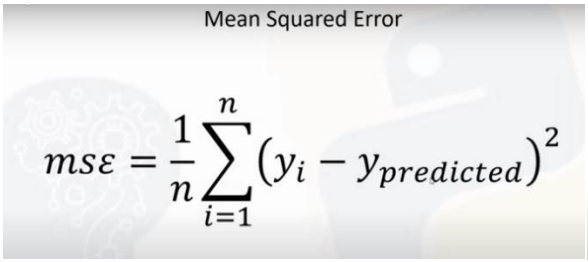
El algoritmo de gradient descent nos permite crear un modelo predictivo al reducir el error (MSE) de un modelo predictivo inicial (propuesto por nosotros) hasta que ese error sea el más bajo posible.

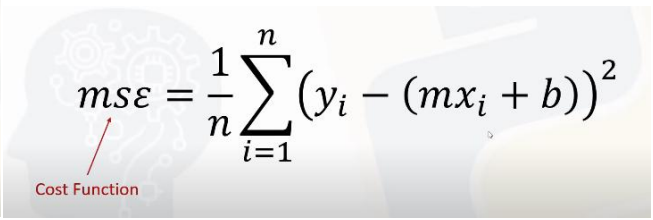
Nota: en ocasiones no es posible llegar a un MSE de 0, esto se puede deber a tener datos insuficientes (ya sea que falten considerar variables, o hacer más mediciones). Cuando esto sucede es probable que no conozcamos uno o más factores que impactan a nuestra variable dependiente.

La siguiente imagen del blog de Krystian Safjan ("Comprehensive Guide to Interpreting R\xB2, MSE, and RMSE for Regression Models."). Muestra visualmente como se ve este MSE. La línea recta es un modelo de predicción y mientras más cerca estén los puntos (valores reales de la variable independiente), menor será el área naranja (MSE).



De forma matemática el mean squared error se calcula de la siguiente manera (imagenes sacada del video Machine Learning Tutorial Python - 4: Gradient Descent and Cost Function):





Esto es de forma literal, restarle cada Y que predice nuestro modelo a cada Y real del set

de datos, elevar cada resta al cuadrado (evita que los valores negativos cancelen a los positivos), sumar los resultados y promediar el total de dicha suma.

Lo que queremos es ver cómo cada una de nuestros coeficientes "m" y nuestro bias "b" afectan este mean squared error. Para ello derivamos MSE con respecto a cada "m" y al "b".

$$\partial/\partial m = \frac{2}{n}\sum_{i=1}^{n} -x_i\left(y_i - (mx_i + b)\right)$$

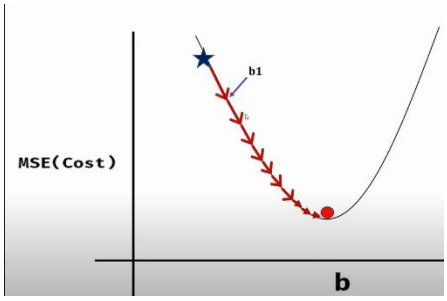
$$\partial/\partial b = \frac{2}{n}\sum_{i=1}^{n} -\left(y_i - (mx_i + b)\right)$$

Imagen sacada del video Machine Learning Tutorial Python - 4: Gradient Descent and Cost Function.

Esta derivada nos dice la dirección a la que cada bias y coeficiente está moviendo nuestro MSE, la idea es irnos moviendo hacia el MSE más pequeño posible.
Esto se consigue utilizando un step (que es lo mismo que el learning rate), que nos dice qué tanto movernos hacia la dirección que hace que el MSE disminuya.

$$m = m - \text{learning rate} * \partial/\partial m$$

$$b = b - \text{learning rate} * \partial/\partial b$$



Imagenes tomadas del video Machine Learning Tutorial Python - 4: Gradient Descent and Cost Function.

En la imagen anterior podemos ver como el valor de b se va actualizando en cada iteración obteniendo un MSE cada vez más pequeño.

El algoritmo de gradient descent suele requerir de varias épocas o iteraciones para llegar a un buen resultado.

Elaboración de código:

Estoy empleando numpy simplemente para manejar los datos en matrices y facilitar el manejo de estos para hacer producto punto entre estas. En otras palabras, para poder multiplicar cada coeficiente de nuestras variables independientes por su respectivo valor y con eso obtener una predicción. Si el código de Python no funciona al ejecutarse, es probable que falte hacer pip install de numpy.

1. El primer paso es identificar y separar nuestra variable dependiente (Y) de las variables independientes (nuestras X) y cargarlas.

En el caso del dataset de abalone, el valor que queremos conseguir es el número de anillos que tiene el árbol (con eso se determina la edad). Además el dataset incluye la columna de Sex que es una variable no cuantitativa por lo que se requiere hacer un hot encoding para poder procesar el dato.

```python
def load_data(filename="abalone.data"):
    data = []
    with open(filename, "r") as f:
        for line in f:
            #separar valores por comas
            values = line.strip().split(",")
            data.append(values)
    #convertir las lineas de datos en un array
    return np.array(data)
```

Carga el archivo.

```python
# Hot encoding para "Sex" (Male, Female, Infant)
def encode_sex(data):
    encoded = []
    for row in data:
        sex = row[0] #sex es el primer valor de la fila
        if sex == "M":
            encoded.append([1, 0, 0])  # M
        elif sex == "F":
            encoded.append([0, 1, 0])  # F
        else:
            encoded.append([0, 0, 1])  # I

    return np.array(encoded)
```

Convierte de cualitativo a cuantitativo.

```python
# Separar variables independientes (X) y dependiente (y)
def preprocess_data(raw_data):
    # Separar columna sex (cualitativa) y el resto
    sex_encoded = encode_sex(raw_data)

    # Convertir todo a float para calculos
    numeric_data = raw_data[:, 1:].astype(float)

    # Nuestra x son las independientes sin el numero de rings (y)
    #hstack permite juntar las 3 columnas del sex_encoded con el resto
    X = np.hstack((sex_encoded, numeric_data[:, :-1]))

    # Variable dependiente = rings
    y = numeric_data[:, -1]
    return X, y
```

Separa la variable dependiente y junta todas las independientes en una sola matriz.

2. Darles un valor inicial a nuestros pesos (los coeficientes de cada una de nuestras variables independientes) y a nuestro bias.

Para esto estoy creando un vector de coeficientes, un coeficiente por cada variable independiente, todos inicializados en 0.

```python
n_samples, n_features = X.shape  # filas, columnas

# Inicializar coeficientes (w) y bias (b)
w = np.zeros(n_features)
b = 0
```

n_samples son las filas y n_features, las columnas.

3. Saco el cálculo de la predicción con el producto punto.

```python
# Prediccion del modelo
y_pred = np.dot(X, w) + b    # X*w + b
```

$(mx_i + b)$

En esta línea se está multiplicando cada fila de X (cada fila tiene n columnas, una por cada variable independiente) por su respectivo coeficiente dentro del vector de pesos w y luego le suma b.

Lo que obtiene la variable y_pred es un arreglo de predicciones del mismo tamaño que las filas de nuestro set de datos. Es decir, si tenemos 15 filas en el dataset, habrá 15 predicciones dentro de y_pred.

4. Sacamos el MSE con la predicción recién calculada:

```
# Calcular costo (MSE)
cost = (1/n_samples) * np.sum((y - y_pred) ** 2)
```

$$mse = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_{predicted})^2$$

A cada valor real de y se le resta el valor de la predicción y se eleva al cuadrado (esto nos deja un arreglo de resultados). Con np.sum sumamos cada valor del arreglo para tener un solo valor numérico el cual se divide entre el número de filas para saber el promedio (MSE).

5. Sacamos un vector de derivadas para saber cuánto y cómo (+/-) influye cada variable independiente y el bias a nuestra variable dependiente.

```
# Derivadas parciales
dw = -(2/n_samples) * np.dot(X.T, (y - y_pred))  # vector de derivadas
db = -(2/n_samples) * np.sum(y - y_pred)
```

$$\partial/\partial m = \frac{2}{n}\sum_{i=1}^{n} -x_i\left(y_i - (mx_i + b)\right)$$

$$\partial/\partial b = \frac{2}{n}\sum_{i=1}^{n} -\left(y_i - (mx_i + b)\right)$$

dw nos muestra un vector (usa np.dot) con la magnitud y signo de la derivada respecto a cada variable independiente. Db (usa.sum) nos da únicamente un valor el cual representa el impacto del bias en el MSE.

6. Utilizando un learning rate actualizamos el valor de nuestro bias y nuestros coeficientes.

```
def gradient_descent(X, y, learning_rate=0.001, iterations=25000):
```

Aquí uso un learning rate default de .001.

```
# Actualizar pesos
w -= learning_rate * dw
b -= learning_rate * db
```

$$m = m - \text{learning rate} * {}^{\partial}/_{\partial m}$$

$$b = b - \text{learning rate} * {}^{\partial}/_{\partial b}$$

7. Finalmente repetimos este proceso las veces que queramos actualizando cada vez los valores del bias y los pesos.

```
def gradient_descent(X, y, learning_rate=0.001, iterations=25000):
```

El default está colocado en 25000.

```
for i in range(iterations):
    # Prediccion del modelo
    y_pred = np.dot(X, w) + b   # X*w + b

    # Calcular costo (MSE)
    cost = (1/n_samples) * np.sum((y - y_pred) ** 2)

    # Derivadas parciales
    dw = -(2/n_samples) * np.dot(X.T, (y - y_pred))  # vector de derivadas
    db = -(2/n_samples) * np.sum(y - y_pred)

    # Actualizar pesos
    w -= learning_rate * dw
    b -= learning_rate * db

    # Mostrar progreso cada cierto número de iteraciones
    if i % 500 == 0:
        print(f"Iteración {i}: w={w}, b={b:.4f}, cost={cost:.4f}")
```

8. Una vez concluidas las épocas regresamos el vector de pesos y el bias final.

```
return w, b
```

```
w, b = gradient_descent(X, y, learning_rate=0.001, iterations=20000)
print("Pesos finales:", w)
print("Bias final:", b)
```

En la foto anterior uso “X” y “y” pero en la aplicación real uso el “X” y “y” de mi dataset de entrenamiento.

Ya que tenemos los valores podemos comenzar a hacer predicciones.

```
# Predecir con los coeficientes obtenidos
def predict(X, w, b):
    return np.dot(X, w) + b
```

9. Crear un modelo funcional.

Para ver que tanto se ajusta a nuevos valores, separamos nuestro dataset en dos, uno para entrenar y otro para probar.

```
# Separar en entrenamiento (80%) y test (20%)
def train_test_split(X, y, test_size=0.2, seed=42):
    np.random.seed(seed)
    n_samples = X.shape[0]
    indices = np.arange(n_samples)
    np.random.shuffle(indices)

    test_size = int(n_samples * test_size)
    test_idx = indices[:test_size]
    train_idx = indices[test_size:]

    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]

    return X_train, X_test, y_train, y_test
```

El código funciona igual solo que en lugar de enviar toda el dataset, enviamos solamente la X y Y de entrenamiento para obtener coeficientes. Después con los coeficientes obtenidos y los valores de X y Y de test podemos comparar los valores que predecimos con valores reales.

10. Predecir y comparar

Con los coeficientes obtenidos y el segmento de prueba comparamos los valores reales con los que pronosticamos.

Estoy utilizando R squared,

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2}$$

Imagen de R-squared in Linear Regression Models: Concepts, Examples.

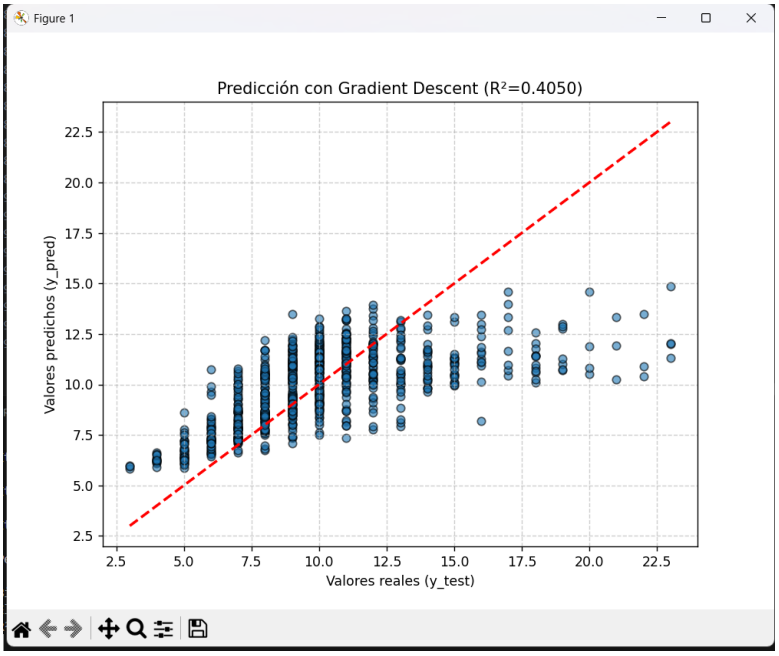
```
# Calcular R squared
def r_squared(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    return 1 - (ss_res / ss_tot)
```

Este valor nos sirve para explicar qué tanto nuestro modelo explica la variación de nuestra variable dependiente. Va de 0 a 1, donde un 1 significa que el modelo es perfecto.

11. Graficar los resultados.

Utilicé matplotlib para mostrar un visual que nos permite entender más fácilmente nuestra predicción contra los valores reales.

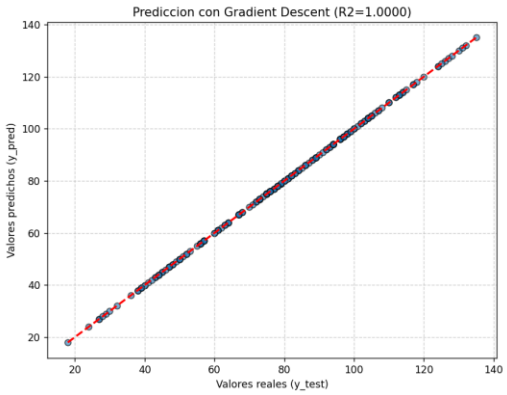
```
plt.figure(figsize=(8,6))
plt.scatter(y_test, y_pred_test, alpha=0.6, edgecolor="k")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', linewidth=2)
plt.xlabel("Valores reales (y_test)")
plt.ylabel("Valores predichos (y_pred)")
plt.title(f"Prediccion con Gradient Descent (R2={r2:.4f})")
plt.grid(True, linestyle="--", alpha=0.6)
plt.show()
```



12. Comprobar los datos y el funcionamiento del código:

Tras ejecutar mi código con el set de datos de abalone, el resultado obtenido es un R2 de .4 lo que significa que mi modelo predice un alrededor de un 40% de la variabilidad. Este valor no es muy alto, esto probablemente se debe a que estamos tratando de hacer una regresión lineal y que otro modelo de regresión (por ejemplo, un exponencial) podría ajustarse mejor al dataset.

Sin embargo, esto no significa que el código no funciona, para demostrar esto creé un dataset artificial con valores estrictamente lineares. Podemos verificar si el código funciona si vemos que los coeficientes son iguales a los que utilicé. Por ser un set de datos artificial, se puede alcanzar un MSE de casi 0, y una R2 de casi 1.




```
Iteración 19500: w=[2.00388771 3.00397674 5.00416822 7.00420067], b=3.9201, cost=0.0006
Pesos finales: [2.00354048 3.00362156 5.00379594 7.00382549]
Bias final: 3.927264157969956
R2 en test set: 1.0000
```

```
np.random.seed(42)
X = np.random.randint(0, 10, (1000, 4))
y = 2*X[:,0] + 3*X[:,1] + 5*X[:,2] + 7*X[:,3] + 4
```