Machine Learning: Gradient descent
Sigmoid function + Clasificación Binaria

Fernando Rizzi

January 14, 2026

Resumen

Anotaciones del libro sobre machine learning, específicamente sobre gradient descent usando la función sigmoide.

1 Generación de Datos Sintéticos

El paquete `sklearn.datasets` nos permite crear puntos en el plano $\mathbb{R}^2$ distribuidos en dos centros distintos:

Listing 1: Código para generar 1000 muestras

```
(X, y) = make_blobs(
    n_samples=1000,
    n_features=2,
    centers=2,
    cluster_std=1.5,
    random_state=1
)
y = y.reshape((y.shape[0],1))
```

- `X` es una matriz 1000×2 con las coordenadas de cada punto.
- `y` contiene la etiqueta (0 o 1). Se convierte en columna (1000×1) para facilitar operaciones matriciales.

Para aplicar el bias trick añadimos una columna de unos:

Listing 2: Agregar la columna de bias

```
X = np.c_[X, np.ones((X.shape[0]))]   # Ahora X es 1000x3
```

Así cada fila $\mathbf{x}_i$ se vuelve: $\mathbf{x}_i = \begin{bmatrix} x_{i1} & x_{i2} & 1 \end{bmatrix}^T$.

2 Función Sigmoide y su Derivada

La sigmoide es la función activación clásica en redes neuronales binarias:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

En código:

Listing 3: Sigmoid

```
def sigmoid_activation(x):
    return 1/(1+np.exp(-x))
```

Para el gradirnt descent necesitamos la derivada de $\sigma(z)$ con respecto a z. Se puede demostrar que:

$$\sigma'(z) = \sigma(z)\,(1 - \sigma(z)).$$

El código implementa esto asumiendo que ya se ha calculado $\sigma(z)$:

Listing 4: Derivada de la sigmoide

```
def sigmoid_deriv(x):
    return x*(1-x)
```

Es decir, si $\hat{y} = \sigma(z)$, entonces

$$\frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y})$$

3 Predicción Binaria

Para convertir la salida continua en una etiqueta binaria usamos un umbral de 0.5:

Listing 5: Función predict

```
def predict(X, W):
    preds = sigmoid_activation(X.dot(W))
    preds[preds <= 0.5] = 0
    preds[preds > 0.5]   = 1
    return preds
```

Donde: $\hat{\mathbf{y}} = \sigma(\mathbf{XW})$, con $\mathbf{W}$ es el vector de pesos (3×1).

4 División del Conjunto

El dataset se separa aleatoriamente en entrenamiento y prueba 50/50:

Listing 6: Split

```
(trainX, testX, trainY, testY) = train_test_split(
    X, y, test_size=0.5, random_state=42)
```

Así tenemos 500 ejemplos de entrenamiento y 500 de prueba.

5 Entrenamiento con gradient descent

Inicializamos los pesos aleatoriamente:

Para cada época calculamos:

1. $\hat{\mathbf{y}} = \sigma(\mathbf{X}_{train}\mathbf{W})$.
2. Error: $\mathbf{e} = \hat{\mathbf{y}} - \mathbf{Y}_{train}$.
3. Pérdida (MSE): $L = \sum e^2$ (se almacena para trazado).
4. Gradiente de la pérdida con respecto a los pesos: $\nabla_W L = \mathbf{X}_{train}^T \left(\mathbf{e} \odot \sigma'(\hat{\mathbf{y}})\right)$, donde $\odot$ indica producto elemento a elemento.
5. Actualizamos los pesos:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \, \nabla_W L,$$

con $\alpha = 0.01$.

El bucle completo:

Listing 7: Loop de entrenamiento

```
for epoch in np.arange(0, args["epochs"]):
    preds = sigmoid_activation(trainX.dot(W))
    error = preds - trainY
    loss  = np.sum(error**2)
    losses.append(loss)

    d        = error * sigmoid_deriv(preds)
    gradient = trainX.T.dot(d)

    W += -args["alpha"] * gradient

    if epoch==0 or (epoch+1)%5==0:
        print("[FER:] epoch = {}, loss={:.7f}".format(int(
            epoch+1), loss))
```

Con 100 épocas el modelo converge muy rápido y la pérdida se aproxima a cero.

6 Evaluación

Después del entrenamiento, usamos predict sobre los datos de prueba y calculamos métricas:

Listing 8: Evaluación

```
preds = predict(testX, W)
print(classification_report(testY, preds))
```

El reporte que se muestra es:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	250
1	1.00	1.00	1.00	250
accuracy	1.00	-	-	500
macro avg	1.00	1.00	1.00	500
weighted avg	1.00	1.00	1.00	500

Esto indica que el modelo clasifica 100% perfecto todos los puntos de prueba. pero esto es porque una recta linear puede separar los valores.

7 Interpretación Matemática del Modelo Final

Al final del entrenamiento, los pesos $\mathbf{W} = [w_1, w_2, b]^T$ definen la recta de decisión: $w_1x+w_2y+b = 0$. El signo de cada punto respecto a esta recta determina su clase. Como los datos fueron generados con dos centros bien separados, una recta lineal basta para separar perfectamente las clases.