Campus 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)

Juan Pablo Castañeda Serrano A01752030

Nombre del Dataset: Estudiantes
Enlace al Dataset: Datos presentes en data.py

Descripción del Dataset:

Cantidad de registros/muestras: 40
Número de características: 2 (Horas de estudio y Horas de dormir)
Número de clases de salida: 2 (Pasó: 1, Reprobó: 0)

```python
def GetData():
# Horas de estudio, Horas de dormir, Resultado (Paso: 1, Reprobo: 0)
    data = [
        [2, 6, 0],
        [3, 5, 0],
        [4, 8, 0],
        [5, 2, 0],
        [6, 5, 0],
        [7, 8, 1],
        [8, 3, 1],
        [9, 7, 1],
        [10, 5, 1],
        [11, 8, 1],
        [3, 7, 0],
        [4, 6, 0],
        [5, 7, 0],
        [6, 3, 0],
        [7, 6, 0],
        [8, 7, 1],
        [9, 4, 1],
        [10, 6, 1],
        [11, 7, 1],
        [12, 8, 1],
        [2, 6, 0],
        [3, 5, 0],
        [4, 8, 0],
        [5, 2, 0],
        [6, 5, 0],
        [7, 8, 1],
        [8, 3, 1],
        [9, 7, 1],
        [10, 5, 1],
        [11, 8, 1],
        [3, 7, 0],
        [4, 6, 0],
        [5, 7, 0],
        [6, 3, 0],
```

Tipo de problema a resolver:

Este es un problema de clasificación binaria. El objetivo es predecir si un estudiante pasa (1) o reprueba (0) basado en las horas de estudio y las horas de dormir.

Descripción de las métricas de desempeño para el subconjunto de entrenamiento:
El script utiliza la función de costo logística regularizada para medir qué tan bien está prediciendo el modelo durante el proceso de entrenamiento. El histórico de esta métrica se guarda en cost_history a lo largo de las iteraciones del gradiente descendente.

```python
def gradient_descent(X, y, theta, alpha, num_iterations, lambda_):
    m = len(y)
    cost_history = []

    for _ in range(num_iterations):
        gradient = (1/m) * np.dot(X.T, (sigmoid(np.dot(X, theta)) - y))
        gradient[1:] += (lambda_ / m) * theta[1:]  # Regularization for theta[1:]
        theta -= alpha * gradient
        cost_history.append(cost(X, y, theta, lambda_))

    return theta, cost_history
```

Descripción de las métricas de desempeño para el subconjunto de prueba:
El script utiliza la métrica de exactitud (Accuracy) para medir la eficacia del modelo en el conjunto de datos. La exactitud es el porcentaje de predicciones correctas respecto al total de predicciones.

```python
def acc(predictions, actual):
    correct = sum(p == a for p, a in zip(predictions, actual))
    return (correct / len(actual)) * 100
```

Predicciones realizadas con el modelo entrenado:
El modelo realiza predicciones en el conjunto de datos utilizando la función predict(). Las predicciones se comparan luego con los valores reales y se muestra la exactitud del modelo.

```
PS C:\Users\sickp\Documents\GitHub\PortafolioImplementacion\Final\Machine Learning\1er entregable corregido> python logistic.py
Predicciones: [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1]
Valores reales: [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

Comparación entre las predicciones generadas y los valores que debieron obtenerse:
El script imprime tanto las predicciones realizadas por el modelo (predictions) como los valores reales (y.tolist()). El usuario puede visualizar estas dos listas para comparar las predicciones del modelo con los valores reales y determinar qué registros fueron predichos correctamente o incorrectamente.

```
Accuracy: 90.00%
```

Partes importantes del código(Leer comentarios):

```python
import numpy as np
from Data import GetData

#Agarra datos
xy=GetData()

X=xy[0]
y=xy[1]


#Funcion sigmoidal
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

#Escalamiento
def feature_scaling(data):
    mean = np.mean(data, axis=0)
    std = np.std(data, axis=0)
    return (data - mean) / std

#Funcion de costo
def cost(X, y, theta, lambda_):
    m = len(y)
    h = sigmoid(np.dot(X, theta))
    reg_term = (lambda_ / (2 * m)) * np.sum(np.square(theta[1:]))
    return (-1/m) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h)) + reg_term
```

```python
#Gradiente descendiente
def gradient_descent(X, y, theta, alpha, num_iterations, lambda_):
    m = len(y)
    cost_history = []

    for _ in range(num_iterations):
        gradient = (1/m) * np.dot(X.T, (sigmoid(np.dot(X, theta)) - y))
        gradient[1:] += (lambda_ / m) * theta[1:]  # Regularization for theta[1:]
        theta -= alpha * gradient
        cost_history.append(cost(X, y, theta, lambda_))

    return theta, cost_history

#Predicción
def predict(X, theta):
    h = sigmoid(np.dot(X, theta))
    return [1 if i >= 0.5 else 0 for i in h]

#Efectividad de prediccion
def acc(predictions, actual):
    correct = sum(p == a for p, a in zip(predictions, actual))
    return (correct / len(actual)) * 100

#Bias
X = np.hstack((np.ones((X.shape[0], 1)), X))
```

```python
#Parametros
theta = np.zeros(X.shape[1])
alpha = 0.01
num_iterations = 10000
lambda_ = 0.1  # Regularization parameter

#Entrenamiento

theta, cost_history = gradient_descent(X, y, theta, alpha, num_iterations, lambda_)



if __name__=="__main__":
    predictions = predict(X, theta)
    print("Predicciones:", predictions)
    print("Valores reales:", y.tolist())
    acc = acc(predictions, y)
    print(f"Accuracy: {acc:.2f}%")
```