

ALAN ALBERTO MOTA YESCAS - A01753924

✓ Pima Indians Diabetes Database

El dataset "Pima Indians Diabetes Database" proviene originalmente del Instituto Nacional de Diabetes y Enfermedades Digestivas y Renales, y su propósito es predecir de manera diagnóstica si un paciente tiene diabetes, basándose en ciertas mediciones diagnósticas. Este conjunto de datos es particularmente relevante porque todos los pacientes son mujeres mayores de 21 años de herencia Pima, un grupo étnico con una alta prevalencia de diabetes tipo 2. El dataset incluye varias variables predictoras médicas, como el número de embarazos, el índice de masa corporal (BMI), el nivel de insulina, la edad, entre otras, y una variable objetivo que indica si el paciente tiene diabetes o no (Outcome). En este proyecto, se te pide implementar manualmente un algoritmo de aprendizaje automático, sin utilizar bibliotecas o frameworks existentes, con el objetivo de predecir la presencia de diabetes en pacientes utilizando este conjunto de datos. La implementación final debe ser capaz de correr independientemente en un archivo de código, sin depender de un entorno de desarrollo interactivo (IDE) o un notebook.

Como se mencionó anteriormente, se utilizan ciertas librerías para la implementación de este modelo, de los cuales son: pandas y numpy. Las librerías importadas juegan un papel crucial en diversas etapas del desarrollo del modelo de Machine Learning. Pandas es fundamental para la manipulación y análisis de los datos, permitiendo cargar, limpiar, y explorar el dataset de manera eficiente. Numpy es esencial para realizar operaciones matemáticas y manejar arreglos numéricos, lo que facilita las manipulaciones algebraicas necesarias durante la implementación del algoritmo. Plotly Graph Objects se utiliza para la visualización de datos, en particular para generar gráficos interactivos que permiten explorar y presentar los resultados de manera clara y atractiva. Finalmente, aunque en este proyecto específico no se utilizarán bibliotecas de Machine Learning para la implementación del algoritmo, Sklearn se incluye para la evaluación de modelos mediante la métrica de la curva ROC y el cálculo del AUC, realización de matriz de confusión, herramientas importantes para medir el desempeño del modelo y comparar su efectividad.

```
import pandas as pd
import numpy as np
import plotly.graph_objects as go
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import confusion_matrix
import plotly.figure_factory as ff
```

El código carga el dataset y luego separa las características predictoras (X) de la variable objetivo (y). Específicamente, **X**, mientras que **y** contiene los valores de la última columna, que corresponde a la variable objetivo "Outcome", indicando si el paciente tiene diabetes o no.

```
# Cargar el dataset
ruta_archivo = "/content/diabetes.csv"
df = pd.read_csv(ruta_archivo)

# Separar características (X) y la variable objetivo (y)
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
```

Normalizamos los datos para que las características tengan una escala similar, lo cual es crucial para el correcto funcionamiento del algoritmo de regresión logística.

```
def normalize(X):
    mean = X.mean(axis=0)
    std = X.std(axis=0)
    return (X - mean) / std

X = normalize(X)
```

La función 'sigmoid' calcula la función sigmoide, que toma una entrada z y devuelve un valor entre 0 y 1, utilizando la fórmula $\text{sigmoid}(z) = \frac{1}{1+e^{-z}}$.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

Se definen dos funciones clave para el entrenamiento de un modelo de regresión logística:

compute_cost: Calcula el costo (o función de pérdida) para el modelo de regresión logística dado un conjunto de características X , la variable objetivo y y los parámetros del modelo θ . Utiliza la función sigmoide para predecir los valores (h), y luego calcula el costo promedio utilizando la función de pérdida logarítmica.

compute_gradient: Calcula el gradiente de la función de costo con respecto a los parámetros θ . Este gradiente es utilizado para actualizar los parámetros durante el entrenamiento mediante un algoritmo de optimización como el descenso de gradiente.

```
def compute_cost(X, y, theta):
    m = len(y)
    h = sigmoid(np.dot(X, theta))
    cost = -(1/m) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))
    return cost

def compute_gradient(X, y, theta):
    m = len(y)
    h = sigmoid(np.dot(X, theta))
    gradient = (1/m) * np.dot(X.T, (h - y))
    return gradient
```

Este bloque de código implementa el proceso de entrenamiento de un modelo de regresión logística mediante el algoritmo de descenso de gradiente. La función 'gradient_descent' ajusta iterativamente los parámetros del modelo **theta** para minimizar la función de costo, actualizando **theta** en cada iteración según el gradiente de la función de costo, escalado por una tasa de aprendizaje. Además, se agrega un término de sesgo al conjunto de características **X** mediante la inclusión de un vector de unos, y se inicializan los pesos **theta** en ceros. Se definen la tasa de aprendizaje y el número de iteraciones que determinan cómo y cuántas veces se actualizarán los parámetros durante el entrenamiento. Finalmente, el modelo se entrena llamando a 'gradient_descent', produciendo un conjunto de parámetros ajustados y un historial de costos, que permite evaluar la convergencia del algoritmo y el desempeño del modelo.

```
def gradient_descent(X, y, theta, learning_rate, num_iterations):
    cost_history = []

    for i in range(num_iterations):
        theta -= learning_rate * compute_gradient(X, y, theta)
        cost = compute_cost(X, y, theta)
        cost_history.append(cost)

    return theta, cost_history

# Agregar un vector de unos para el término de sesgo (bias)
X_bias = np.c_[np.ones((X.shape[0], 1)), X]

# Inicializar los pesos (theta)
theta = np.zeros(X_bias.shape[1])

# Definir parámetros
learning_rate = 0.01
num_iterations = 1000

# Entrenar el modelo
theta, cost_history = gradient_descent(X_bias, y, theta, learning_rate, num_iterations)
```

Se implementa la función de predicción para el modelo de regresión logística:

predict: Esta función toma como entrada las características X y los parámetros entrenados θ . Calcula la probabilidad de que cada ejemplo en X pertenezca a la clase positiva (es decir, que el valor de la variable objetivo sea 1) utilizando la función sigmoide aplicada al producto punto de X y θ . Luego, convierte estas probabilidades en predicciones binarias: si la probabilidad es mayor o igual a 0.5, se predice un 1 (indicando la presencia de diabetes), y si es menor, se predice un 0.

y_pred: Se generan predicciones para el conjunto de entrenamiento utilizando la función predict con las características X_{bias} (que incluyen el término de sesgo) y los parámetros ajustados θ obtenidos durante el entrenamiento.


```
def predict(X, theta):
    prob = sigmoid(np.dot(X, theta))
    return [1 if p >= 0.5 else 0 for p in prob]

# Predicciones en el conjunto de entrenamiento
y_pred = predict(X_bias, theta)
```

Se calcula la precisión del modelo de regresión logística, una métrica que indica qué porcentaje de las predicciones del modelo coinciden con las etiquetas reales. La función `accuracy` compara las predicciones generadas por el modelo (`y_pred`) con los valores reales (`y_true`), contando cuántas predicciones fueron correctas y dividiendo este número por el total de ejemplos. Finalmente, se calcula y se imprime la precisión del modelo como un porcentaje, proporcionando una medida clara de qué tan bien ha funcionado el modelo en el conjunto de datos de entrenamiento.

```
def accuracy(y_true, y_pred):
    correct = np.sum(y_true == y_pred)
    return correct / len(y_true)

# Calcular la precisión
acc = accuracy(y, y_pred)
print(f"Precisión del modelo: {acc * 100:.2f}%")
```

 Precisión del modelo: 76.95%

El modelo de regresión logística ajustado ha obtenido una precisión del 76.95% al verificar las predicciones hechas en el conjunto de datos de entrenamiento. Basado en el escenario mencionado anteriormente, tiene sentido que el valor de precisión sea fijo y no difiera con cada ejecución repetida del código, suponiendo que se estén utilizando los mismos datos de entrada y los parámetros e hiperparámetros (por ejemplo, la tasa de aprendizaje y las iteraciones) se inicialicen de la misma manera. Esto se debe a que el proceso de entrenamiento en la regresión logística es determinista en escenarios en los que no se usa aleatoriedad en el proceso, es decir, dados un conjunto de datos y un conjunto de condiciones iniciales, el modelo siempre tomará la misma ruta en el espacio de parámetros para llegar al mismo resultado. Esto es porque no hay variabilidad en la entrada de datos (es decir, no se está usando ninguna técnica de división aleatoria como `train_test_split`), el algoritmo siempre comienza inicializando los pesos del modelo en ceros, y el descenso de gradiente sigue siendo determinista. Es decir, ya que no hay aleatoriedad involucrada en ningún paso del proceso, el resultado final, incluido el rendimiento del modelo, será exactamente el mismo en cada ejecución. Este código es solo un ejemplo muy simple de un modelo de regresión logística que se puede mejorar o modificar según los requisitos, con técnicas que se pueden agregar para mejorar la eficiencia o al crear aleatoriedad si se desea que el resultado sea diferente entre las ejecuciones.

El siguiente gráfico muestra cómo la función de costo disminuye hacia un mínimo a lo largo de las iteraciones a medida que se entrena el modelo de regresión logística. Del mismo modo, se puede deducir que el costo es una función monótona decreciente de las iteraciones, lo que significa que el algoritmo de descenso de gradiente está haciendo un buen trabajo disminuyendo los parámetros del modelo de manera efectiva para minimizar el error. La caída es más aguda al principio, lo que significa que las primeras iteraciones están contribuyendo en gran medida a la mejora de la precisión. Pero luego se nivela, lo que significa que el modelo realmente está llegando cerca de la convergencia. Este es un comportamiento clásico en la optimización de modelos de machine learning, a saber, que una vez que comienza el proceso de descenso, la magnitud de los movimientos iniciales es la más significativa, y los ajustes se refinan progresivamente mientras los parámetros se acercan a su equilibrio. Como se muestra en la gráfica, el modelo está efectivamente descendiendo hacia el mínimo de la función de costo, lo cual es una buena señal de que está aprendiendo correctamente de los datos.

```
# Gráfico de la función de costo
fig = go.Figure()

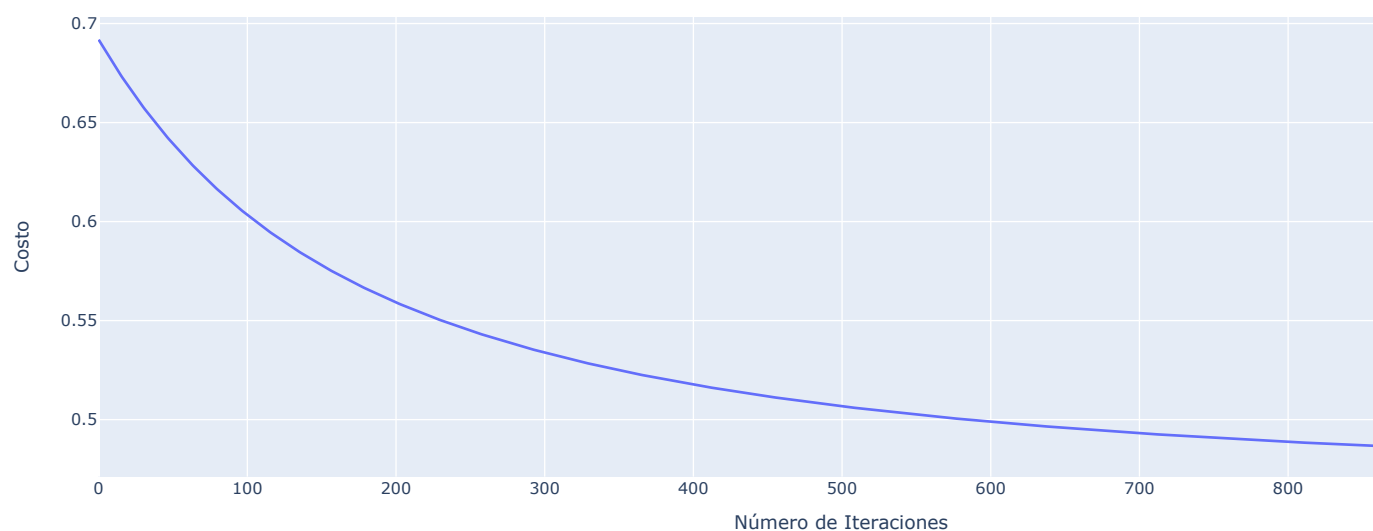
fig.add_trace(go.Scatter(
    x=list(range(len(cost_history))),
    y=cost_history,
    mode='lines',
    name='Costo'
))

fig.update_layout(
    title='Convergencia de la función de costo',
    xaxis_title='Número de Iteraciones',
    yaxis_title='Costo'
)

fig.show()
```



Convergencia de la función de costo



Lo siguiente es un gráfico de la ROC (Característica de Operación del Receptor), que mide el modelo de regresión logística por cuán buena es la capacidad de discriminación entre las clases positiva y negativa (diabetes vs. no diabetes). El eje x representa la tasa de falsos positivos (FPR) y el eje y representa la tasa de verdaderos positivos (TPR). La línea azul ilustra cuánto cambia la TPR con la FPR, dado que el umbral del modelo también cambia. El mejor modelo posible estaría a lo largo del punto más alto en la esquina superior izquierda, donde la TPR es alta y la FPR es lo más baja posible. La línea roja que diagonalmente atraviesa el gráfico es un modelo que elige al azar, con un AUC (Área Bajo la Curva) de 0.5. Aquí la AUC es 0.84, lo cual es bueno, y el modelo tiene la capacidad de discriminar correctamente las clases la mayor parte del tiempo. Mientras más cerca esté la curva ROC de la esquina superior izquierda, mejor será el rendimiento del modelo en términos de sensibilidad y especificidad.

```

# Calcular las probabilidades de predicción
y_prob = sigmoid(np.dot(X_bias, theta))

# Calcular los valores de la curva ROC
fpr, tpr, thresholds = roc_curve(y, y_prob)
roc_auc = auc(fpr, tpr)

# Gráfico de la curva ROC
fig = go.Figure()

fig.add_trace(go.Scatter(
    x=fpr,
    y=tpr,
    mode='lines',
    name=f'Curva ROC (AUC = {roc_auc:.2f})'
))

fig.add_trace(go.Scatter(
    x=[0, 1],
    y=[0, 1],
    mode='lines',
    line=dict(dash='dash', color='red'),
    name='Línea de referencia'
))

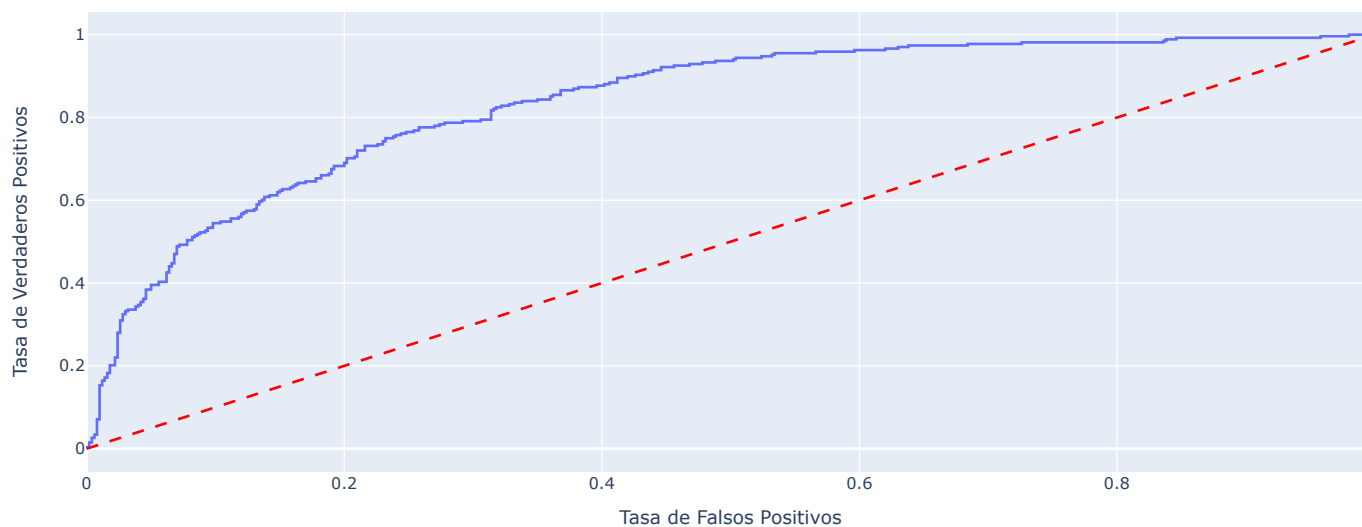
fig.update_layout(
    title='Curva ROC',
    xaxis_title='Tasa de Falsos Positivos',
    yaxis_title='Tasa de Verdaderos Positivos'
)

fig.show()

```



Curva ROC



La siguiente matriz de confusión muestra un análisis más detallado del rendimiento del modelo de regresión logística al clasificar la información en dos categorías discretas: "diabetes" (1) y "no diabetes" (0). Cada uno de los valores en la matriz es el recuento de ejemplos clasificados por el modelo en dicha clase o combinación de clases:

- 115 ejemplos fueron clasificados correctamente como "diabetes" (Verdaderos Positivos).
- 62 ejemplos fueron clasificados correctamente como "no diabetes" (Verdaderos Negativos).
- 153 ejemplos que se clasificaron incorrectamente como "no diabetes" cuando tenían diabetes (Falsos Negativos).
- 438 ejemplos que se clasificaron incorrectamente como "diabetes" cuando no tenían diabetes (Falsos Positivos).

La matriz establece que el modelo tiende a arrojar muchos falsos positivos, especialmente sobre los ejemplos que no tienen diabetes (muchos falsos positivos). Esto muestra que el modelo es un poco sesgado hacia la predicción de la clase positiva y el sesgo debe ser corregido de

alguna manera mediante la corrección del umbral o algún otro ajuste para obtener una mejor precisión y equilibrio entre las clases. La matriz de confusión es una excelente manera de entender la debilidad y la fortaleza de un modelo en las predicciones.

```
# Calcular la matriz de confusión
cm = confusion_matrix(y, y_pred)

# Crear una matriz de confusión con plotly
z = cm.tolist()
x = ['Predicho: 0', 'Predicho: 1']
y = ['Real: 0', 'Real: 1']

fig = ff.create_annotated_heatmap(
    z,
    x=x,
    y=y,
    colorscale='Blues',
    showscale=True
)

fig.update_layout(
    title='Matriz de Confusión',
    xaxis_title='Predicción',
    yaxis_title='Real'
)

fig.show()
```

