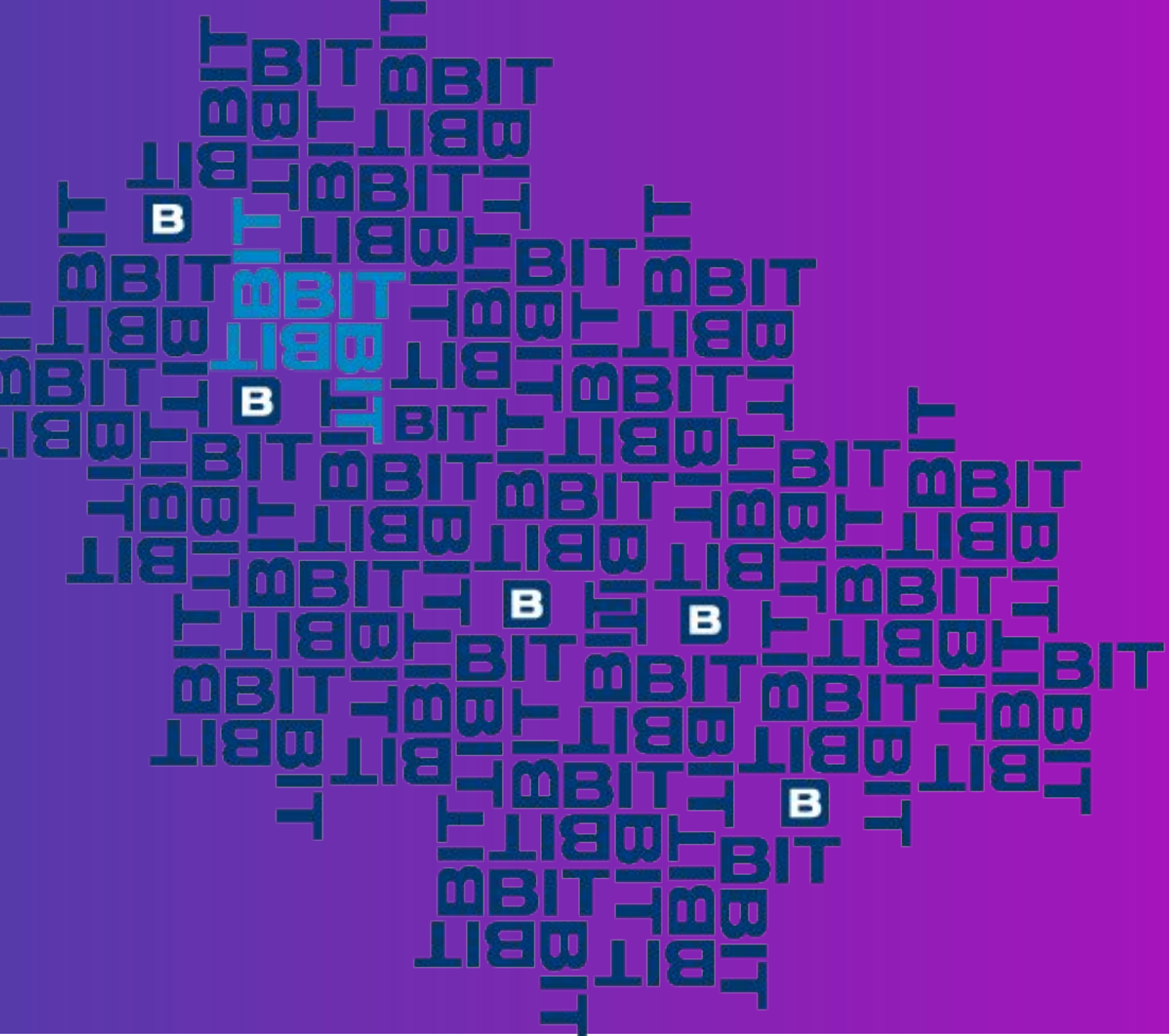
BIT BUILD INNOVATE TRANSFORM

Bootcamp IA
Módulo 2.
Machine Learning
-Métricas

METRICAS DE MEDICIÓN DE DESEMPEÑO



BIT BUILD INNOVATE TRANSFORM

Evaluación del Modelo

Métricas Regresión Lineal

BIT BUILD INNOVATE TRANSFORM

Error Cuadrático Medio (MSE - Mean Squared Error)

El **Error Cuadrático Medio** mide el promedio de los cuadrados de los errores o diferencias entre los valores predichos y los valores reales. Penaliza los errores grandes más fuertemente que los errores pequeños.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Donde:

- $y^{(i)}$ es el valor real.
- $\hat{y}^{(i)}$ es el valor predicho por el modelo.
- n es el número total de observaciones.

Interpretación: Un valor de MSE más bajo indica que las predicciones están más cerca de los valores reales.

```python
def mean_squared_error(y_true, y_pred):
    mse = np.mean((y_true - y_pred) ** 2)
    return mse

# Suponemos que tienes tus predicciones y los valores reales de y
y_pred = np.dot(X, theta_opt)  # Predicciones usando los parámetros optimizados
mse = mean_squared_error(y, y_pred)
print(f"Error Cuadrático Medio (MSE): {mse:.4f}")
```

BIT BUILD INNOVATE TRANSFORM

Error Cuadrático Medio de Predicción (MSEP - Mean Squared Prediction Error)

El **MSEP** es una métrica similar al MSE, pero se utiliza para evaluar el error de predicción en un conjunto de datos de prueba que no se utilizó durante el entrenamiento. Esto nos da una idea de qué tan bien funcionará el modelo en datos no vistos.

$$MSEP = \frac{1}{n} \sum_{i=1}^{n} \left(y_{\text{test}}^{(i)} - \hat{y}_{\text{test}}^{(i)} \right)^2$$

Interpretación: Un valor bajo de MSEP indica que el modelo generaliza bien a datos no vistos.

BIT BUILD INNOVATE TRANSFORM

Raíz del Error Cuadrático Medio (RMSE)

```python
def root_mean_squared_error(y_true, y_pred):
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    return rmse

# Calcular RMSE
rmse = root_mean_squared_error(y, y_pred)
print(f"Raíz del Error Cuadrático Medio (RMSE): {rmse:.4f}")
```

BIT BUILD INNOVATE TRANSFORM

Error Absoluto Medio (MAE - Mean Absolute Error)

El **Error Absoluto Medio** mide el promedio de las diferencias absolutas entre los valores reales y los valores predichos. A diferencia del MSE, no penaliza tanto los errores grandes, ya que no eleva los errores al cuadrado.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

Interpretación: Un valor de MAE bajo significa que, en promedio, los errores absolutos son pequeños.

```python
def mean_absolute_error(y_true, y_pred):
    mae = np.mean(np.abs(y_true - y_pred))
    return mae

# Calcular MAE
mae = mean_absolute_error(y, y_pred)
print(f"Error Absoluto Medio (MAE): {mae:.4f}")
```

BIT BUILD INNOVATE TRANSFORM

R^2 o Coeficiente de Determinación

El R^2 mide qué tan bien se ajustan las predicciones a los valores reales. Va desde 0 hasta 1, donde:

- **1** indica que el modelo explica perfectamente la variabilidad de los datos.
- **0** indica que el modelo no explica nada de la variabilidad de los datos.

La fórmula del R^2 es:

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

```python
def r2_score(y_true, y_pred):
    ss_total = np.sum((y_true - np.mean(y_true)) ** 2)
    ss_residual = np.sum((y_true - y_pred) ** 2)
    r2 = 1 - (ss_residual / ss_total)
    return r2

# Calcular R^2
r2 = r2_score(y, y_pred)
print(f"Coeficiente de Determinación (R^2): {r2:.4f}")
```

Donde:

- $\hat{y}^{(i)}$ es el valor predicho.
- $y^{(i)}$ es el valor real.
- $\bar{y}$ es el valor promedio de los valores reales.

Interpretación:

- $R^2 = 1$: Predicciones perfectas (ajuste perfecto).
- $R^2 = 0$: El modelo no explica ninguna variación en los datos.
- $R^2 < 0$: El modelo es peor que simplemente predecir el promedio de los datos.

BIT BUILD INNOVATE TRANSFORM

R2 Ajustado (Adjusted R^2)

El R^2 **ajustado** es una versión del R^2 que ajusta el valor según el número de predictores en el modelo. A medida que se añaden más variables al modelo, el R^2 tiende a aumentar, incluso si esas variables no aportan mucho al ajuste. El R^2 ajustado corrige esta tendencia.

$$R^2_{\text{ajustado}} = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

Donde:

- n es el número total de observaciones.
- k es el número de predictores.

Interpretación: Un R^2 ajustado más alto indica un mejor ajuste del modelo a los datos, teniendo en cuenta la complejidad del modelo.

```python
def adjusted_r2_score(y_true, y_pred, n_features):
    n = len(y_true)  # Número de ejemplos
    r2 = r2_score(y_true, y_pred)
    adj_r2 = 1 - (1 - r2) * (n - 1) / (n - n_features - 1)
    return adj_r2

# Calcular R^2 Ajustado
n_features = X.shape[1]  # Número de características
adj_r2 = adjusted_r2_score(y, y_pred, n_features)
print(f"Coeficiente de Determinación Ajustado (R^2 ajustado): {adj_r2:.4f}")
```

Errores Residuos

El **error residual** es la diferencia entre los valores reales y los valores predichos por el modelo. Los residuos se utilizan para identificar posibles patrones no capturados por el modelo.

$$\text{Residuo} = y^{(i)} - \hat{y}^{(i)}$$

- **Análisis gráfico de los residuos**: Es común graficar los residuos para asegurarse de que no hay patrones sistemáticos en los errores. Un buen modelo tendrá residuos distribuidos aleatoriamente alrededor de 0.

```
# Calcular los residuales
residuals = y - y_pred

# Graficar los residuales
plt.scatter(y_pred, residuals)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel('Valores Predichos')
plt.ylabel('Residuales')
plt.title('Gráfico de los Residuales')
plt.grid(True)
plt.show()
```

Métricas Clasificación

BIT BUILD INNOVATE TRANSFORM

Exactitud (Accuracy)

La **precisión** es la métrica más básica y mide el porcentaje de predicciones correctas, es decir, cuántas veces el modelo predice correctamente si un dato pertenece a la clase positiva o negativa.

$$\text{Precisión} = \frac{\text{Número de predicciones correctas}}{\text{Total de predicciones}} = \frac{TP + TN}{TP + TN + FP + FN}$$

Donde:

- **TP**: Verdaderos positivos (True Positives)
- **TN**: Verdaderos negativos (True Negatives)
- **FP**: Falsos positivos (False Positives)
- **FN**: Falsos negativos (False Negatives)

```python
# Función para calcular la precisión
def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred) * 100

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

Limitación: La precisión puede ser engañosa en conjuntos de datos desbalanceados, donde una clase domina, haciendo que el modelo parezca más preciso de lo que realmente es.

Ejercicio básico de accuracy

```
Etiquetas Reales [1 1]
Etiquetas Predichas [0 1]
```

	Clase Predicha Positiva	Clase Predicha Negativa
Clase Verdadera Positiva	TP	FN
Clase Verdadera Negativa	FP	TN

Ejercicio básico de accuracy

```
Etiquetas Reales [1 0 1 0]
Etiquetas Predichas [1 1 1 0]
Presicion del modelo: 0.75
```

	Clase Predicha Positiva	Clase Predicha Negativa
Clase Verdadera Positiva	TP	FN
Clase Verdadera Negativa	FP	TN

BIT BUILD INNOVATE TRANSFORM

Precisión (Precision)

La **precisión** mide la proporción de predicciones positivas que son correctas. Es útil cuando los **falsos positivos** son costosos o importantes de minimizar.

$$\text{Precisión} = \frac{TP}{TP + FP}$$

Es decir, de todas las predicciones que fueron positivas, ¿cuántas eran realmente positivas?

```python
# Función para calcular la precisión (precision)
def precision_score(y_true, y_pred):
    TP = np.sum((y_true == 1) & (y_pred == 1))
    FP = np.sum((y_true == 0) & (y_pred == 1))
    return TP / (TP + FP)

# Calcular la precisión (precision)
precision = precision_score(y, predictions)
print(f"Precisión (Precision): {precision:.2f}")
```

La **exhaustividad** o **recall** mide la proporción de verdaderos positivos que fueron correctamente identificados. Es útil cuando es crucial no pasar por alto los **falsos negativos**.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Es decir, de todas las instancias que realmente eran positivas, ¿cuántas fueron correctamente predichas?

```python
# Función para calcular el recall (exhaustividad)
def recall_score(y_true, y_pred):
    TP = np.sum((y_true == 1) & (y_pred == 1))
    FN = np.sum((y_true == 1) & (y_pred == 0))
    return TP / (TP + FN)

# Calcular el recall (sensibilidad)
recall = recall_score(y, predictions)
print(f"Recall: {recall:.2f}")
```

BIT BUILD INNOVATE TRANSFORM

Precisión (Precision)

	Clase Predicha Positiva	Clase Predicha Negativa
Clase Verdadera Positiva	TP = 2	FN = 0
Clase Verdadera Negativa	FP = 1	TN = 1

$$F1 = 2 \cdot \frac{\text{Precisión} \cdot \text{Recall}}{\text{Precisión} + \text{Recall}}$$

La **precisión** mide la proporción de predicciones positivas que son correctas. Es útil cuando los **falsos positivos** son costosos o importantes de minimizar.

$$\text{Precisión} = \frac{TP}{TP + FP}$$

Es decir, de todas las predicciones que fueron positivas, ¿cuántas eran realmente positivas?

La **exhaustividad** o **recall** mide la proporción de verdaderos positivos que fueron correctamente identificados. Es útil cuando es crucial no pasar por alto los **falsos negativos**.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Es decir, de todas las instancias que realmente eran positivas, ¿cuántas fueron correctamente predichas?

BIT BUILD INNOVATE TRANSFORM

Precisión (Precision)

	Clase Predicha Positiva	Clase Predicha Negativa
Clase Verdadera Positiva	TP = 2	FN = 0
Clase Verdadera Negativa	FP = 1	TN = 1

$$F1 = 2 \cdot \frac{\text{Precisión} \cdot \text{Recall}}{\text{Precisión} + \text{Recall}}$$

$$\text{Precisión} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

BIT BUILD INNOVATE TRANSFORM

Exhaustividad (Recall) o Sensibilidad

La **exhaustividad** o **recall** mide la proporción de verdaderos positivos que fueron correctamente identificados. Es útil cuando es crucial no pasar por alto los **falsos negativos**.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Es decir, de todas las instancias que realmente eran positivas, ¿cuántas fueron correctamente predichas?

```python
# Función para calcular el recall (exhaustividad)
def recall_score(y_true, y_pred):
    TP = np.sum((y_true == 1) & (y_pred == 1))
    FN = np.sum((y_true == 1) & (y_pred == 0))
    return TP / (TP + FN)

# Calcular el recall (sensibilidad)
recall = recall_score(y, predictions)
print(f"Recall: {recall:.2f}")
```

BIT BUILD INNOVATE TRANSFORM

F1-Score

El **F1-Score** es la media armónica entre la **precisión** y el **recall**, y se utiliza para encontrar un equilibrio entre estas dos métricas. Es útil cuando existe un trade-off entre falsos positivos y falsos negativos.

$$F1 = 2 \cdot \frac{\text{Precisión} \cdot \text{Recall}}{\text{Precisión} + \text{Recall}}$$

Un F1-Score más alto significa que el modelo tiene un buen equilibrio entre precisión y recall.

```python
# Función para calcular el F1-Score
def f1_score(y_true, y_pred):
    prec = precision_score(y_true, y_pred)
    rec = recall_score(y_true, y_pred)
    return 2 * (prec * rec) / (prec + rec)

# Calcular el F1-Score
f1 = f1_score(y, predictions)
print(f"F1-Score: {f1:.2f}")
```

BIT BUILD INNOVATE TRANSFORM

Matriz de Confusión

La **matriz de confusión** es una tabla que organiza las predicciones correctas e incorrectas en función de las clases verdaderas y predichas. Es útil para obtener una visión general del rendimiento del modelo.

	Clase Predicha Positiva	Clase Predicha Negativa
Clase Verdadera Positiva	TP	FN
Clase Verdadera Negativa	FP	TN

La matriz de confusión muestra el número de verdaderos positivos, verdaderos negativos, falsos positivos y falsos negativos, y proporciona una visión más detallada que la precisión sola.

```python
# Función para calcular la matriz de confusión
def confusion_matrix(y_true, y_pred):
    TP = np.sum((y_true == 1) & (y_pred == 1))
    TN = np.sum((y_true == 0) & (y_pred == 0))
    FP = np.sum((y_true == 0) & (y_pred == 1))
    FN = np.sum((y_true == 1) & (y_pred == 0))

    return np.array([[TP, FN], [FP, TN]])

# Calcular la matriz de confusión
conf_matrix = confusion_matrix(y, predictions)
print("Matriz de Confusión:")
print(conf_matrix)
```

```
Matriz de Confusión:
[[TP  FN]
 [FP  TN]]
```

BIT BUILD INNOVATE TRANSFORM

Tasa de Verdaderos Positivos (TPR)
Tasa de Falsos Positivos (FPR)

La **Tasa de Verdaderos Positivos (TPR)** o **sensibilidad** es la fracción de ejemplos positivos que el modelo predice correctamente. Ya se mencionó antes como **recall**.

$$\text{TPR} = \frac{TP}{TP + FN}$$

La **Tasa de Falsos Positivos (FPR)** mide la proporción de predicciones incorrectas entre los ejemplos que son realmente negativos.

$$\text{FPR} = \frac{FP}{FP + TN}$$

La **curva ROC** muestra el rendimiento del modelo en términos de la tasa de verdaderos positivos (recall) contra la tasa de falsos positivos para varios umbrales. El **AUC (Área bajo la curva ROC)** mide la capacidad del modelo para separar las clases.

```python
from sklearn.metrics import roc_curve, auc

# Calcular las probabilidades de la clase positiva
y_probs = sigmoid(np.dot(X, theta_opt))

# Calcular la curva ROC
fpr, tpr, thresholds = roc_curve(y, y_probs)

# Calcular el AUC
roc_auc = auc(fpr, tpr)

# Graficar la curva ROC
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Tasa de Falsos Positivos (FPR)')
plt.ylabel('Tasa de Verdaderos Positivos (TPR)')
plt.title('Curva ROC')
plt.legend(loc="lower right")
plt.show()
```

Curva ROC y AUC (Área bajo la curva ROC)

- La **curva ROC (Receiver Operating Characteristic)** es una gráfica que muestra la **tasa de verdaderos positivos** (recall o sensibilidad) frente a la **tasa de falsos positivos** para diferentes umbrales de decisión.
- El **AUC (Área bajo la curva ROC)** mide el área bajo la curva ROC. Un AUC de 1.0 indica un modelo perfecto, mientras que un AUC de 0.5 indica un modelo que no tiene poder predictivo (equivalente a una clasificación aleatoria).

$$AUC = \int_0^1 \text{TPR} \cdot d(\text{FPR})$$

Donde:

- **TPR**: True Positive Rate (tasa de verdaderos positivos, o recall)
- **FPR**: False Positive Rate (tasa de falsos positivos).

Un modelo con un AUC cercano a 1 es considerado un buen modelo.

BIT BUILD INNOVATE TRANSFORM

Log-Loss (Pérdida Logarítmica)

La **log-loss** o **pérdida logarítmica** mide la incertidumbre de las predicciones del modelo. Penaliza los errores de clasificación en función de la probabilidad predicha para cada clase. Un valor de log-loss más bajo indica mejores predicciones.

$$\text{Log-Loss} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right)$$

Donde:

- y_i es el valor real (0 o 1).
- $\hat{y}_i$ es la probabilidad predicha por el modelo para la clase positiva.

```python
# Función para calcular Log-Loss (Pérdida Logarítmica)
def log_loss(y_true, y_probs):
    m = len(y_true)
    return -(1/m) * np.sum(y_true * np.log(y_probs) + (1 - y_true) * np.log(1 - y_probs))

# Calcular Log-Loss
log_loss_value = log_loss(y, y_probs)
print(f"Log-Loss: {log_loss_value:.4f}")
```

BIT
BUILD
INNOVATE
TRANSFORM

¡Gracias!