Instituto Tecnológico y de Estudios Superiores de Monterrey

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

Módulo 2:

Análisis y Reporte sobre el desempeño del modelo

Alumna:

Mariluz Daniela Sánchez Morales

Profesor:

Jorge Adolfo Ramírez Uresti

11 de Septiembre de 2024

Modelo Elegido: Regresión logística

Train, Test, Validation

En este modelo se han dividido los datos en 3 sets diferentes con diferentes porcentajes del dataset original:

- **Entrenamiento**: Se utilizó el 80% de los datos para ajustar los parámetros del modelo y minimizar el error de las predicciones.
- **Validación:** Se uso un 20% de los datos obtenidos del entrenamiento para seleccionar los mejores hiperparámetros y que de esta forma no se produzca un sobreajuste del modelo
- **Prueba:** Se utilizo el 20% restante del dataset original para evaluar el rendimiento del modelo con datos que no vio anteriormente. Nota: De igual forma se generó otro dataset con valores distintos para probar con más datos el modelo de regresión logística.

```
# Manejo de datos
samples = np.c_[np.ones(len(samples)), samples]
samples = scale_features(samples)

data_size = len(samples)

# Los datos se revuelven para posteriormente dividirlos equitativamente
indices = np.arange(data_size)
np.random.shuffle(indices)

# Rangos de partición de los datos
train_size = int(0.8 * data_size)
validation_size = int(0.2 * train_size)
test_size = data_size - train_size

# División de datos en train, validación y test
train_indices = indices[:train_size]
val_indices = indices[train_size:train_size + validation_size]
test_indices = indices[train_size + validation_size:]

# División de datos por set de datos
X_train, Y_train = [samples[i] for i in train_indices], [y[i] for i in train_indices]
X_val, Y_val = [samples[i] for i in val_indices], [y[i] for i in val_indices]
X_test, Y_test = [samples[i] for i in test_indices], [y[i] for i in test_indices]

# Entrenar modelo X_train, Y_train, alpha, "Train"
alpha = 0.3
params, errorTrain, errorVal = logistic_regression(X_train,Y_train,X_val,Y_val, alpha)
graph = plot_errors(errorTrain, errorVal)
```

Imagen 1. División del dataset de entrenamiento.

```
""" PRUEBA: Nuevo set de datos """

# Nuevos datos para testeo
X_test_nuevos = [1500.45, 3200.67, 4500.12, 2800.40, 3700.89,
                 5000.00, 1900.55, 3300.75, 2100.20, 2600.30,
                 3400.80, 2200.90, 2800.10, 3600.70, 4000.50,
                 2900.60, 3400.20, 2400.85, 3100.30, 3300.40,
                 2200.45, 3100.50, 2700.70, 3500.60, 4300.40,
                 2500.20, 3300.10, 1900.75, 3000.80, 3100.60,
                 2000.65, 3700.10, 3100.70, 3200.20, 2500.90,
                 2900.30, 4100.70, 3000.60, 2300.55, 2500.30]

y_test_nuevos = [0, 1, 1, 0, 1,
                 1, 0, 1, 0, 0,
                 1, 0, 0, 1, 1,
                 0, 1, 0, 1, 1,
                 0, 1, 0, 1, 1,
                 0, 1, 0, 1, 1,
                 0, 1, 1, 1, 0,
                 0, 1, 1, 0, 0]

# Preparar datos: agregar columna de bias y escalar características
X_test_nuevos = np.c_[np.ones(len(X_test_nuevos)), X_test_nuevos]  #
X_test_nuevos = scale_features(X_test_nuevos)  # Normalizar
```

Imagen 2. Dataset de prueba 100% de datos

Después de la separación de los datos, se evaluó el modelo inicial con un learning rate de 0.3 y con normalización Min-Max. Los resultados del error en entrenamiento y validación fueron los siguientes:

- **Error en el Conjunto de Entrenamiento**:
  - Antes de la normalización: 0.0923
- **Error en el Conjunto de Validación**:
  - Antes de la normalización: 0.0649

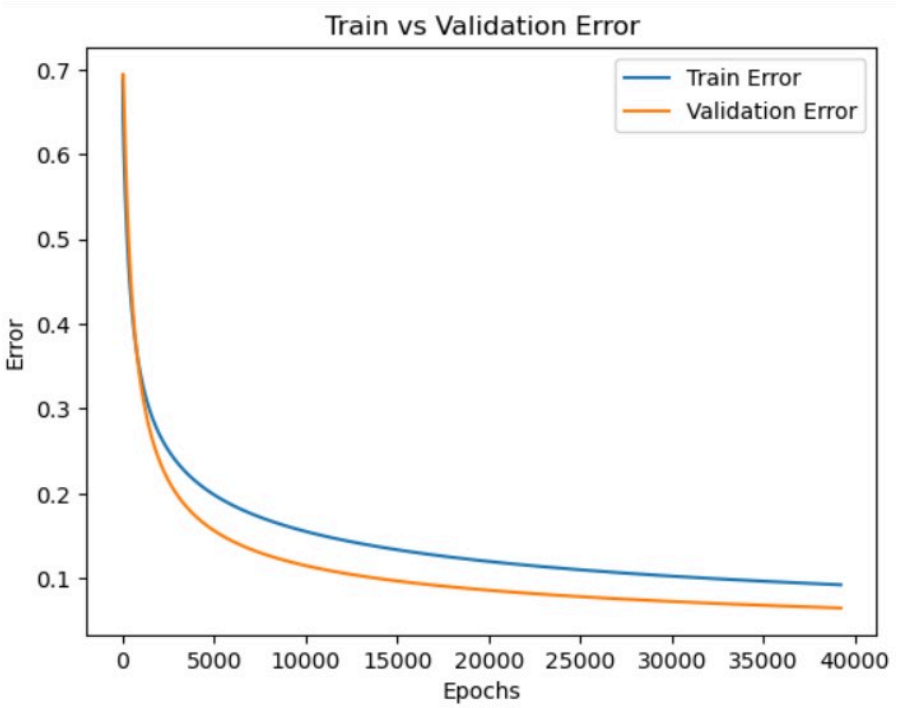

Imagen 3. Gráfica de error con normalización Min-Max

Sin embargo se hizo un ajuste en la normalización de datos donde se uso la desviación estándar y el resultado de este cambio mejoró ligeramente el error del modelo:

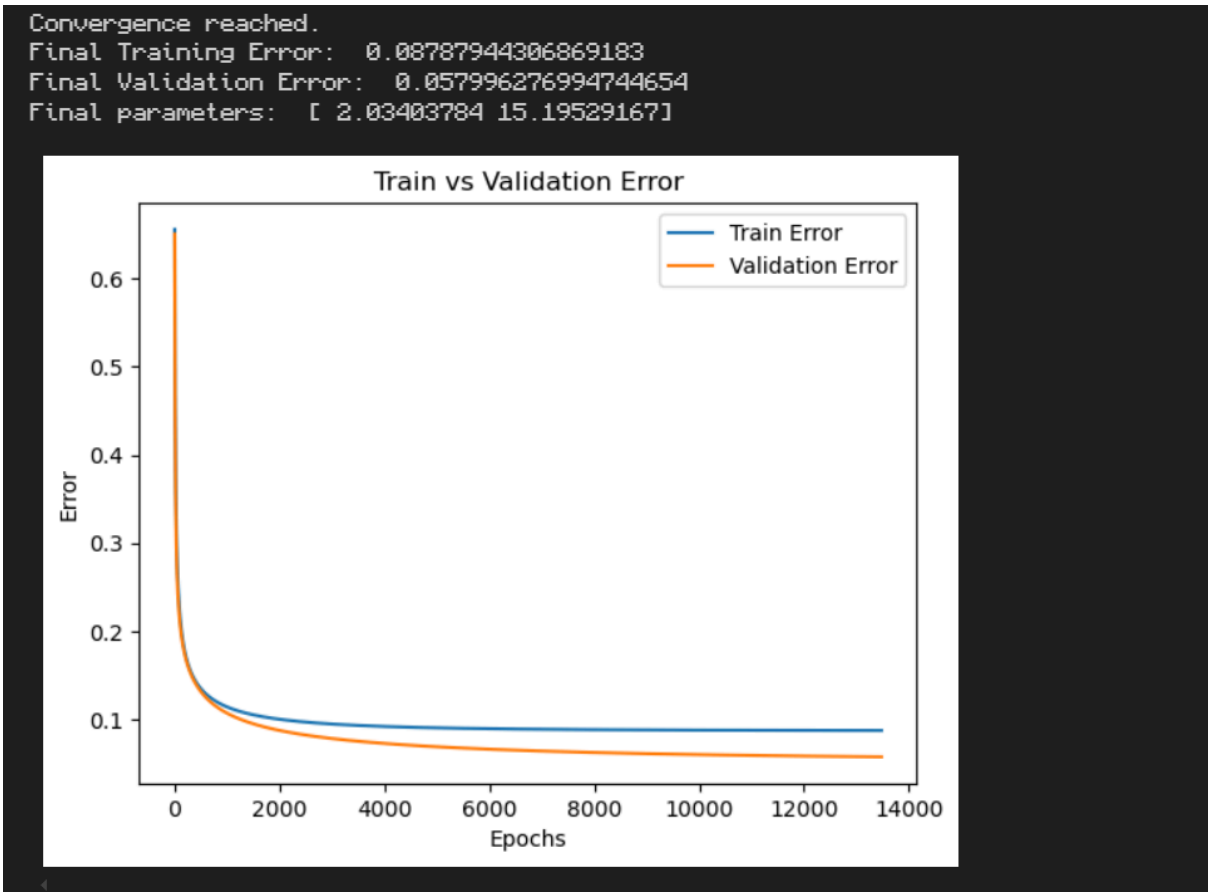

Imagen 4. Gráfica de error con desviación estándar

Bias

El **sesgo** en este modelo parece ser **bajo**.

- El error en el conjunto de entrenamiento es pequeño tanto antes como después del cambio en el tipo de normalización. Esto quiere decir que el modelo se ajusta bien a las relaciones encontradas en el dataset lo que evita el underfitting del modelo.
- Este cambio realizado en la normalización reafirmo que el modelo se ajusta correctamente a los datos debido a la disminución en el error de entrenamiento

Varianza

El **grado de varianza** también parece ser **bajo**:

- Antes de la normalización, la diferencia entre el error de entrenamiento y el error de validación era pequeña, lo que ya es una señal positiva de baja varianza.
- Después de la normalización, la diferencia se redujo aún más (0.0879 vs. 0.0580), lo que indica que el modelo ahora generaliza mejor a nuevos datos.
- Sin embargo, la diferencia entre ambos sigue siendo de 0.03 aproximadamente pero definitivamente mejoró la generalización.

Nivel de ajuste del modelo

Nivel de ajuste del modelo: **Good Fit**

- Tanto antes como después del cambio de normalización, el ajuste del modelo era bueno sin embargo, en la segunda grafica, se destaca la mejora del modelo ya que hay un ligero decrecimiento en la diferencia de errores entre entrenamiento y validación lo que sugiere que se ha alcanzado un nivel adecuado de ajuste a los datos.

Regularización y Ajuste de Parámetros

Para mejorar aún más el desempeño del modelo y prevenir cualquier posible sobreajuste en casos futuros, se puede aplicar **regularización**. La regularización ayuda a controlar la magnitud de los parámetros del modelo, evitando que algunos pesos se vuelvan demasiado grandes, lo que podría causar sobreajuste.

```python
def compute_cost_with_regularization(params, samples, y, lambda_):
    """ Cálculo de costo con regularización L2 """
    m = len(y)
    hyp = h(params, samples)

    epsilon = 1e-10
    hyp = np.where(hyp == 0, epsilon, hyp)
    hyp = np.where(hyp == 1, 1 - epsilon, hyp)

    cost = -np.mean(y * np.log(hyp) + (1 - y) * np.log(1 - hyp))

    # Añadir el término de regularización (L2)
    regularization = (lambda_ / (2 * m)) * np.sum(params[1:] ** 2)
    return cost + regularization


def gradient_descent_with_regularization(params, samples, y, alpha, lambda_):
    m = len(y)
    hyp = h(params, samples)
    gradient = np.dot(samples.T, (hyp - y)) / m

    # Añadir regularización
    params[1:] -= (alpha * (gradient[1:] + (lambda_ / m) * params[1:]))
    params[0] -= alpha * gradient[0]  # No regularizamos el término independiente

    return params
```

Imagen 5. Aplicación de la regularización

Resultados después de aplicar regularización:

- **Error en el Conjunto de Entrenamiento**:
  - Antes de la regularización: 0.0879
  - Después de la regularización: 0.0892 (ligeramente mayor debido a la penalización)
- **Error en el Conjunto de Validación**:
  - Antes de la regularización: 0.0580
  - Después de la regularización: 0.0543 (mejoró en validación)

Impacto de la Regularización:

- **Bias (Sesgo)**: El sesgo se mantiene bajo, ya que el modelo sigue ajustando bien los datos.
- **Varianza**: La regularización ha reducido la varianza al evitar que algunos parámetros sean demasiado grandes, mejorando así la capacidad del modelo para generalizar.
- **Ajuste del Modelo**: El ajuste es ahora aún más controlado y robusto. El modelo sigue estando en un **good fit**, pero con una mayor capacidad para evitar el sobreajuste.