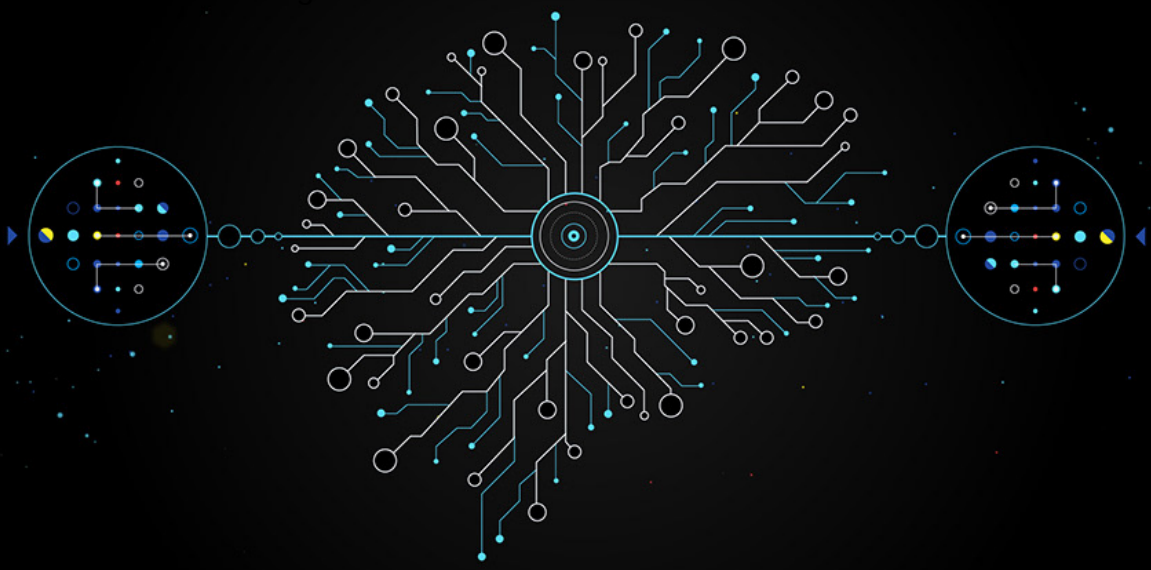
# Exploración y Desarrollo de Modelos de Machine Learning

Ingresa texto



Por Antonio Richaud

# Exploración y desarrollo de modelos de machine learning

**Autor:** Antonio Richaud

## Descripción

En este notebook, exploraremos y desarrollaremos varios modelos de aprendizaje automático para abordar diferentes tipos de problemas utilizando diversos conjuntos de datos. Comenzaremos con un Modelo Perceptrón para la clasificación de dígitos escritos a mano, seguido por un Modelo de Regresión lineal para predecir el valor medio de viviendas en California. Posteriormente, implementaremos un Modelo de Perceptrón Multicapa (MultiLayer Perceptron, MLP) o Red Neuronal Profunda (Deep Neural Network, DNN) para resolver un problema de regresión. Finalmente, abordaremos un Problema de Clasificación utilizando el conjunto de datos del Titanic para predecir la supervivencia de los pasajeros. A lo largo de este notebook, se detallarán los pasos necesarios para la preparación de los datos, la construcción de los modelos, su entrenamiento y evaluación, proporcionando una comprensión integral de las técnicas y prácticas en el aprendizaje automático.

### 1. Modelo Perceptrón

En este modelo, se implementó y evaluó un Perceptrón utilizando el conjunto de datos de dígitos de sklearn. Primero, se importaron las bibliotecas necesarias (numpy, pandas y sklearn) y se cargaron los datos de dígitos, que consisten en imágenes de 8x8 píxeles representando dígitos escritos a mano, junto con sus etiquetas correspondientes. Luego, se dividieron los datos en conjuntos de entrenamiento y prueba en una proporción de 75% y 25%, respectivamente. Posteriormente, se construyó un modelo de Perceptrón utilizando la clase Perceptron de sklearn.linear_model y se entrenó con los datos de entrenamiento. Para evaluar el rendimiento del modelo, se calcularon las precisiones en los conjuntos de entrenamiento y prueba, obteniendo valores de aproximadamente 96.6% y 92.7%, respectivamente. Además, se generaron y analizaron reportes de clasificación que detallan métricas como precisión, recall y f1-score para cada clase de dígito, demostrando que el modelo tiene un buen rendimiento general y es capaz de clasificar correctamente la mayoría de las imágenes de dígitos tanto en el conjunto de entrenamiento como en el de prueba.

[ ]:
```
import numpy as np
import pandas as pd
import sklearn
from sklearn.datasets import load_digits
```

#### Cargar el conjunto de datos de dígitos

[ ]:
```
digits = load_digits()
digits
```

[ ]:
```
{'data': array([[ 0.,  0.,  5., …,  0.,  0.,  0.],
        [ 0.,  0.,  0., …, 10.,  0.,  0.],
        [ 0.,  0.,  0., …, 16.,  9.,  0.],
        …,
        [ 0.,  0.,  1., …,  6.,  0.,  0.],
        [ 0.,  0.,  2., …, 12.,  0.,  0.],
        [ 0.,  0., 10., …, 12.,  1.,  0.]]),
 'target': array([0, 1, 2, …, 8, 9, 8]),
 'frame': None,
 'feature_names': ['pixel_0_0',
  'pixel_0_1',
  'pixel_0_2',
  'pixel_0_3',
  'pixel_0_4',
  'pixel_0_5',
  'pixel_0_6',
  'pixel_0_7',
  'pixel_1_0',
  'pixel_1_1',
  'pixel_1_2',
  'pixel_1_3',
  'pixel_1_4',
  'pixel_1_5',
  'pixel_1_6',
  'pixel_1_7',
  'pixel_2_0',
  'pixel_2_1',
  'pixel_2_2',
  'pixel_2_3',
  'pixel_2_4',
  'pixel_2_5',
  'pixel_2_6',
  'pixel_2_7',
  'pixel_3_0',
  'pixel_3_1',
  'pixel_3_2',
  'pixel_3_3',
  'pixel_3_4',
  'pixel_3_5',
  'pixel_3_6',
  'pixel_3_7',
  'pixel_4_0',
  'pixel_4_1',
  'pixel_4_2',
```

```
 'pixel_4_3',
 'pixel_4_4',
 'pixel_4_5',
 'pixel_4_6',
 'pixel_4_7',
 'pixel_5_0',
 'pixel_5_1',
 'pixel_5_2',
 'pixel_5_3',
 'pixel_5_4',
 'pixel_5_5',
 'pixel_5_6',
 'pixel_5_7',
 'pixel_6_0',
 'pixel_6_1',
 'pixel_6_2',
 'pixel_6_3',
 'pixel_6_4',
 'pixel_6_5',
 'pixel_6_6',
 'pixel_6_7',
 'pixel_7_0',
 'pixel_7_1',
 'pixel_7_2',
 'pixel_7_3',
 'pixel_7_4',
 'pixel_7_5',
 'pixel_7_6',
 'pixel_7_7'],
'target_names': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
'images': array([[[ 0.,  0.,  5., …,  1.,  0.,  0.],
        [ 0.,  0., 13., …, 15.,  5.,  0.],
        [ 0.,  3., 15., …, 11.,  8.,  0.],
        …,
        [ 0.,  4., 11., …, 12.,  7.,  0.],
        [ 0.,  2., 14., …, 12.,  0.,  0.],
        [ 0.,  0.,  6., …,  0.,  0.,  0.]],

       [[ 0.,  0.,  0., …,  5.,  0.,  0.],
        [ 0.,  0.,  0., …,  9.,  0.,  0.],
        [ 0.,  0.,  3., …,  6.,  0.,  0.],
        …,
        [ 0.,  0.,  1., …,  6.,  0.,  0.],
        [ 0.,  0.,  1., …,  6.,  0.,  0.],
        [ 0.,  0.,  0., …, 10.,  0.,  0.]],

       [[ 0.,  0.,  0., …, 12.,  0.,  0.],
```

```
        [ 0.,  0.,  3., …, 14.,  0.,  0.],
        [ 0.,  0.,  8., …, 16.,  0.,  0.],
        …,
        [ 0.,  9., 16., …,  0.,  0.,  0.],
        [ 0.,  3., 13., …, 11.,  5.,  0.],
        [ 0.,  0.,  0., …, 16.,  9.,  0.]],

       …,

       [[ 0.,  0.,  1., …,  1.,  0.,  0.],
        [ 0.,  0., 13., …,  2.,  1.,  0.],
        [ 0.,  0., 16., …, 16.,  5.,  0.],
        …,
        [ 0.,  0., 16., …, 15.,  0.,  0.],
        [ 0.,  0., 15., …, 16.,  0.,  0.],
        [ 0.,  0.,  2., …,  6.,  0.,  0.]],

       [[ 0.,  0.,  2., …,  0.,  0.,  0.],
        [ 0.,  0., 14., …, 15.,  1.,  0.],
        [ 0.,  4., 16., …, 16.,  7.,  0.],
        …,
        [ 0.,  0.,  0., …, 16.,  2.,  0.],
        [ 0.,  0.,  4., …, 16.,  2.,  0.],
        [ 0.,  0.,  5., …, 12.,  0.,  0.]],

       [[ 0.,  0., 10., …,  1.,  0.,  0.],
        [ 0.,  2., 16., …,  1.,  0.,  0.],
        [ 0.,  0., 15., …, 15.,  0.,  0.],
        …,
        [ 0.,  4., 16., …, 16.,  6.,  0.],
        [ 0.,  8., 16., …, 16.,  8.,  0.],
        [ 0.,  1.,  8., …, 12.,  1.,  0.]]]),
 'DESCR': ".. _digits_dataset:\n\nOptical recognition of handwritten digits
dataset\n--------------------------------------------------\n\n**Data Set
Characteristics:**\n\n    :Number of Instances: 1797\n    :Number of Attributes:
64\n    :Attribute Information: 8x8 image of integer pixels in the range
0..16.\n    :Missing Attribute Values: None\n    :Creator: E. Alpaydin (alpaydin
'@' boun.edu.tr)\n    :Date: July; 1998\n\nThis is a copy of the test set of the
UCI ML hand-written digits datasets\nhttps://archive.ics.uci.edu/ml/datasets/Opt
ical+Recognition+of+Handwritten+Digits\n\nThe data set contains images of hand-
written digits: 10 classes where\neach class refers to a digit.\n\nPreprocessing
programs made available by NIST were used to extract\nnormalized bitmaps of
handwritten digits from a preprinted form. From a\ntotal of 43 people, 30
contributed to the training set and different 13\nto the test set. 32x32 bitmaps
are divided into nonoverlapping blocks of\n4x4 and the number of on pixels are
counted in each block. This generates\nan input matrix of 8x8 where each element
is an integer in the range\n0..16. This reduces dimensionality and gives
```

```
invariance to small\ndistortions.\n\nFor info on NIST preprocessing routines,
see M. D. Garris, J. L. Blue, G.\nT. Candela, D. L. Dimmick, J. Geist, P. J.
Grother, S. A. Janet, and C.\nL. Wilson, NIST Form-Based Handprint Recognition
System, NISTIR 5469,\n1994.\n\n|details-start|\n**References**\n|details-
split|\n\n- C. Kaynak (1995) Methods of Combining Multiple Classifiers and
Their\n  Applications to Handwritten Digit Recognition, MSc Thesis, Institute
of\n  Graduate Studies in Science and Engineering, Bogazici University.\n- E.
Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.\n- Ken Tang and
Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin.\n  Linear
dimensionalityreduction using relevance weighted LDA. School of\n  Electrical
and Electronic Engineering Nanyang Technological University.\n  2005.\n- Claudio
Gentile. A New Approximate Maximal Margin Classification\n  Algorithm. NIPS.
2000.\n\n|details-end|"}
```

### Visualización y Preprocesamiento de los Datos

[ ]:
```
# Muestra los datos en forma de matriz.

digits['data']
```

[ ]:
```
array([[ 0.,  0.,  5., …,  0.,  0.,  0.],
       [ 0.,  0.,  0., …, 10.,  0.,  0.],
       [ 0.,  0.,  0., …, 16.,  9.,  0.],
       …,
       [ 0.,  0.,  1., …,  6.,  0.,  0.],
       [ 0.,  0.,  2., …, 12.,  0.,  0.],
       [ 0.,  0., 10., …, 12.,  1.,  0.]])
```

[ ]:
```
# Muestra las etiquetas de los datos.

digits['target']
```

[ ]:
```
array([0, 1, 2, …, 8, 9, 8])
```

[ ]:
```
# x se asigna a los datos de las imágenes, y y a las etiquetas.

x = digits.data
y = digits.target
```

### Normalización de los datos

[ ]:
```
# Esta línea normaliza los valores de píxeles dividiendo por 255.0
# Esto puede mejorar el rendimiento del modelo.
# x = x / 255.0

# División del conjunto de datos en entrenamiento y prueba
# Importar la función train_test_split para dividir los datos
from sklearn.model_selection import train_test_split
```

```
# Dividir los datos en conjuntos de entrenamiento y prueba
# El 25% de los datos se usarán para pruebas y el 75% para entrenamiento.
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25,␣
 ↪random_state=0)
```

### Construcción del modelo de perceptrón

[ ]:
```
# Importamos el modelo Perceptron desde sklearn.linear_model
from sklearn.linear_model import Perceptron

# Creamos una instancia del modelo Perceptron
perceptron_model = Perceptron()

# Entrenamos el modelo con los datos de entrenamiento
# fit(x_train, y_train) ajusta el modelo a los datos de entrenamiento
perceptron_model.fit(x_train, y_train)
```

[ ]:
```
Perceptron()
```

### Evaluación del Modelo de Perceptrón

[ ]:
```
# Evaluar la precisión del modelo en el conjunto de entrenamiento y prueba
print("Cuál es la precisión en el conjunto de entrenamiento del modelo␣
 ↪Perceptron:")
print(perceptron_model.score(x_train, y_train))

print("Cuál es la precisión en el conjunto de prueba del modelo Perceptron:")
print(perceptron_model.score(x_test, y_test))

# Generar predicciones para los conjuntos de entrenamiento y prueba
predicted_train = perceptron_model.predict(x_train)
predicted_test = perceptron_model.predict(x_test)

# Importar las métricas de evaluación
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix

# Calcular y mostrar la precisión en el conjunto de entrenamiento y prueba
print("Precisión en el conjunto de entrenamiento:", accuracy_score(y_train,␣
 ↪predicted_train))
print("Precisión en el conjunto de prueba:", accuracy_score(y_test,␣
 ↪predicted_test))

# Mostrar el reporte de clasificación para el conjunto de entrenamiento
print("Reporte de clasificación para el conjunto de entrenamiento")
print(classification_report(y_train, predicted_train))
```

```python
# Mostrar el reporte de clasificación para el conjunto de prueba
print("Reporte de clasificación para el conjunto de prueba")
print(classification_report(y_test, predicted_test))
```

```
Cuál es la precisión en el conjunto de entrenamiento del modelo Perceptron:
0.9658500371195249
Cuál es la precisión en el conjunto de prueba del modelo Perceptron:
0.9266666666666666
Precisión en el conjunto de entrenamiento: 0.9658500371195249
Precisión en el conjunto de prueba: 0.9266666666666666
Reporte de clasificación para el conjunto de entrenamiento
              precision    recall  f1-score   support

           0       1.00      0.99      1.00       141
           1       0.96      0.88      0.92       139
           2       1.00      0.98      0.99       133
           3       0.92      1.00      0.96       138
           4       0.95      0.99      0.97       143
           5       1.00      0.97      0.98       134
           6       0.92      1.00      0.96       129
           7       1.00      0.98      0.99       131
           8       0.99      0.87      0.93       126
           9       0.94      0.99      0.96       133

    accuracy                           0.97      1347
   macro avg       0.97      0.97      0.97      1347
weighted avg       0.97      0.97      0.97      1347

Reporte de clasificación para el conjunto de prueba
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        37
           1       0.94      0.74      0.83        43
           2       0.95      0.91      0.93        44
           3       0.85      0.98      0.91        45
           4       0.88      1.00      0.94        38
           5       0.98      0.96      0.97        48
           6       0.84      1.00      0.91        52
           7       1.00      0.94      0.97        48
           8       1.00      0.79      0.88        48
           9       0.90      0.96      0.93        47

    accuracy                           0.93       450
   macro avg       0.93      0.93      0.93       450
weighted avg       0.93      0.93      0.93       450
```

**Interpretación de Resultados del Modelo Perceptron  Precisión en el Conjunto de Entrenamiento y Prueba**

*Interpretación:*

- Precisión en el Conjunto de Entrenamiento: 0.9658500371195249
- Significado: El modelo predijo correctamente aproximadamente el 96.6% de las etiquetas en el conjunto de entrenamiento.
- Precisión en el Conjunto de Prueba: 0.9266666666666666
- Significado: El modelo predijo correctamente aproximadamente el 92.7% de las etiquetas en el conjunto de prueba.

La precisión más alta en el conjunto de entrenamiento indica que el modelo se ajusta bien a los datos de entrenamiento. La precisión en el conjunto de prueba es ligeramente menor, lo cual es común y esperado, ya que el modelo no ha visto estos datos durante el entrenamiento.

**Reporte de Clasificación para el Conjunto de Entrenamiento**

*Interpretación:*

- precision: La precisión es la proporción de verdaderos positivos entre el número total de ejemplos predichos para cada clase.
- Ejemplo: Para la clase 0, la precisión es 1.00, lo que significa que todas las predicciones para la clase 0 son correctas.
- recall: El recall es la proporción de verdaderos positivos entre el número total de ejemplos que realmente pertenecen a cada clase.
- Ejemplo: Para la clase 0, el recall es 0.99, lo que significa que el 99% de los ejemplos reales de la clase 0 fueron correctamente identificados por el modelo.
- f1-score: El f1-score es la media armónica de la precisión y el recall. Es una medida equilibrada que considera tanto falsos positivos como falsos negativos.
- Ejemplo: Para la clase 0, el f1-score es 1.00.
- support: El soporte es el número de ocurrencias reales de la clase en el conjunto de datos.
- Ejemplo: Para la clase 0, hay 141 ocurrencias.

La precisión, el recall y el f1-score altos en la mayoría de las clases indican que el modelo está funcionando bien en los datos de entrenamiento.

**Reporte de Clasificación para el Conjunto de Prueba**

*Interpretación:*

- La precisión, el recall y el f1-score son generalmente altos en el conjunto de prueba, lo que indica que el modelo generaliza bien a datos no vistos.
- La clase 1 tiene una menor precisión y recall en comparación con otras clases, lo que puede indicar que el modelo tiene más dificultad para predecir correctamente los ejemplos de esta clase.
- Las métricas macro avg y weighted avg también son altas, lo que sugiere que el rendimiento del modelo es consistente a través de las diferentes clases.

En general, estos resultados indican que el modelo de perceptrón ha sido bien entrenado y tiene un buen rendimiento tanto en el conjunto de datos de entrenamiento como en el de prueba. Sin embargo, siempre hay espacio para mejorar, por ejemplo, ajustando hiperparámetros, utilizando técnicas de normalización (como la que estaba comentada), o explorando modelos más complejos.

### 2. Modelo de Regresión

Desarrollamos un modelo de regresión lineal utilizando el conjunto de datos de viviendas de California. Comenzamos importando las bibliotecas necesarias y cargando los datos, creando un DataFrame con las características y la etiqueta (valor medio de las viviendas). Luego, verificamos y aseguramos que no hubiera valores nulos en el conjunto de datos. Dividimos los datos en características (X) y etiquetas (y) y normalizamos las características para mejorar el rendimiento del modelo. Dividimos los datos en conjuntos de entrenamiento y prueba en una proporción de 80/20 y construimos un modelo de regresión lineal con Keras, añadiendo capas densas, normalización por lotes y dropout para evitar el sobreajuste. Compilamos el modelo utilizando el optimizador Adam y la función de pérdida de error cuadrático medio, y lo entrenamos durante 100 épocas. Evaluamos el modelo en el conjunto de prueba, obteniendo una pérdida de 0.39. Para analizar los resultados, graficamos la evolución de la pérdida durante el entrenamiento y la validación, y comparamos las predicciones del modelo con los valores reales, mostrando que las predicciones estaban razonablemente cerca de los valores reales. Finalizamos con una gráfica de predicciones frente a valores reales para visualizar el rendimiento del modelo, concluyendo que el modelo tiene un buen desempeño general.

#### Cargar el Conjunto de Datos

[ ]:
```
# Importar las bibliotecas necesarias
import tensorflow as tf
from tensorflow import keras
from keras import Sequential
from keras.layers import Dense, Dropout, BatchNormalization
from sklearn.datasets import fetch_california_housing
import pandas as pd
```

[ ]:
```
# Cargar el conjunto de datos de viviendas de California
housing = fetch_california_housing()

# Crear un DataFrame con los datos
data = pd.DataFrame(housing.data, columns=housing.feature_names)
data['MedHouseVal'] = housing.target

# Mostrar los primeros registros del DataFrame
data.head()
```

[ ]:

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccup | Latitude | \ |
|---|---|---|---|---|---|---|---|---|
| 0 | 8.3252 | 41.0 | 6.984127 | 1.023810 | 322.0 | 2.555556 | 37.88 | |
| 1 | 8.3014 | 21.0 | 6.238137 | 0.971880 | 2401.0 | 2.109842 | 37.86 | |
| 2 | 7.2574 | 52.0 | 8.288136 | 1.073446 | 496.0 | 2.802260 | 37.85 | |
| 3 | 5.6431 | 52.0 | 5.817352 | 1.073059 | 558.0 | 2.547945 | 37.85 | |

```
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude  MedHouseVal
0    -122.23        4.526
1    -122.22        3.585
2    -122.24        3.521
3    -122.25        3.413
4    -122.25        3.422
```

### Creación y Entrenamiento del Modelo de Regresión

[ ]:
```
# Verificar si hay valores nulos en el DataFrame
data.isnull().sum()
```

[ ]:
```
MedInc         0
HouseAge       0
AveRooms       0
AveBedrms      0
Population     0
AveOccup       0
Latitude       0
Longitude      0
MedHouseVal    0
dtype: int64
```

[ ]:
```
# Dividir los datos en características (X) y etiqueta (y)
x = data.iloc[:, :-1]
y = data.iloc[:, -1]

# Mostramos los primeros registros del conjunto de características
x.head()
```

[ ]:
```
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude
0    -122.23
1    -122.22
2    -122.24
3    -122.25
4    -122.25
```

[ ]:
```
# Normalizamos los datos
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x = sc.fit_transform(x)

# Mostramos los primeros registros después de la normalización
pd.DataFrame(x).head()
```

[ ]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 2.344766 | 0.982143 | 0.628559 | -0.153758 | -0.974429 | -0.049597 | 1.052548 | -1.327835 |
| 1 | 2.332238 | -0.607019 | 0.327041 | -0.263336 | 0.861439 | -0.092512 | 1.043185 | -1.322844 |
| 2 | 1.782699 | 1.856182 | 1.155620 | -0.049016 | -0.820777 | -0.025843 | 1.038503 | -1.332827 |
| 3 | 0.932968 | 1.856182 | 0.156966 | -0.049833 | -0.766028 | -0.050329 | 1.038503 | -1.337818 |
| 4 | -0.012881 | 1.856182 | 0.344711 | -0.032906 | -0.759847 | -0.085616 | 1.038503 | -1.337818 |

[ ]:
```
# Dividir los datos en conjuntos de entrenamiento y prueba
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,␣
↪random_state=0)

# Mostrar las dimensiones de los conjuntos de datos resultantes
print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
```

```
(16512, 8) (16512,) (4128, 8) (4128,)
```

El resultado muestra las dimensiones de los conjuntos de datos de entrenamiento y prueba después de la división. Vamos a ver que significa cada parte :

1. (16512, 8): Este es el tamaño del conjunto de datos de entrenamiento (x_train).

- 16512: Número de muestras (instancias) en el conjunto de entrenamiento.
- 8: Número de características (columnas) en cada muestra.

2. (16512,): Este es el tamaño de las etiquetas del conjunto de entrenamiento (y_train).

- 16512: Número de etiquetas correspondientes a las muestras en el conjunto de entrenamiento.

3. (4128, 8): Este es el tamaño del conjunto de datos de prueba (x_test).

- 4128: Número de muestras en el conjunto de prueba.
- 8: Número de características en cada muestra.

4. (4128,): Este es el tamaño de las etiquetas del conjunto de prueba (y_test).

- 4128: Número de etiquetas correspondientes a las muestras en el conjunto de prueba.

**Interpretación:**

- Conjunto de Entrenamiento (x_train y y_train):
- Tiene 16,512 muestras y cada muestra tiene 8 características.
- Hay 16,512 etiquetas correspondientes a estas muestras.
- Conjunto de Prueba (x_test y y_test):
- Tiene 4,128 muestras y cada muestra tiene 8 características.
- Hay 4,128 etiquetas correspondientes a estas muestras.

**División del Conjunto de Datos**

La división de datos con test_size=0.2 significa que el 20% de los datos se utilizan para el conjunto de prueba y el 80% para el conjunto de entrenamiento. Aquí, el conjunto de datos original se ha dividido en 16,512 muestras para el entrenamiento y 4,128 muestras para la prueba, lo cual es consistente con la proporción especificada.

**Construcción y Entrenamiento del Modelo de Regresión (Keras)**

1. Construcción del Modelo:
   - Sequential: Se utiliza para crear un modelo secuencial.
   - Dense: Añade capas densamente conectadas al modelo.
   - BatchNormalization: Normaliza las activaciones de la capa anterior, lo que puede acelerar el entrenamiento y mejorar la estabilidad.
   - Dropout: Apaga aleatoriamente un porcentaje de las neuronas durante el entrenamiento para evitar el sobreajuste.
2. Compilación del Modelo:
   - optimizer='adam': Utiliza el optimizador Adam.
   - loss='mean_squared_error': Utiliza el error cuadrático medio como función de pérdida.
3. Entrenamiento del Modelo:
   - fit: Entrena el modelo con los datos de entrenamiento.
   - validation_split=0.2: Utiliza el 20% de los datos de entrenamiento para validación durante el entrenamiento.
4. Evaluación del Modelo:
   - evaluate: Evalúa el modelo con los datos de prueba y devuelve la pérdida.
5. Predicciones:
   - predict: Realiza predicciones con los datos de prueba.

[ ]:
```
# Construir el modelo de regresión
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=x_train.shape[1]))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(32, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(1, activation='linear'))
```

[ ]:
```
# Compilar el modelo
model.compile(optimizer='adam', loss='mean_squared_error')
```

[ ]:
```
# Entrenar el modelo
history = model.fit(x_train, y_train, epochs=100, validation_split=0.2,␣
↪batch_size=32)
```

```
Epoch 1/100
413/413 [==============================] - 6s 8ms/step - loss: 3.5886 -
val_loss: 0.6063
Epoch 2/100
413/413 [==============================] - 4s 10ms/step - loss: 1.2882 -
val_loss: 0.4706
Epoch 3/100
413/413 [==============================] - 1s 3ms/step - loss: 0.9117 -
val_loss: 0.4339
Epoch 4/100
413/413 [==============================] - 1s 3ms/step - loss: 0.7868 -
val_loss: 0.4406
Epoch 5/100
413/413 [==============================] - 1s 3ms/step - loss: 0.6962 -
val_loss: 0.4573
Epoch 6/100
413/413 [==============================] - 2s 4ms/step - loss: 0.6430 -
val_loss: 0.4610
Epoch 7/100
413/413 [==============================] - 1s 4ms/step - loss: 0.6257 -
val_loss: 0.4773
Epoch 8/100
413/413 [==============================] - 1s 4ms/step - loss: 0.6033 -
val_loss: 0.4598
Epoch 9/100
413/413 [==============================] - 2s 4ms/step - loss: 0.5939 -
val_loss: 0.4381
Epoch 10/100
413/413 [==============================] - 2s 5ms/step - loss: 0.5788 -
val_loss: 0.4466
Epoch 11/100
413/413 [==============================] - 3s 6ms/step - loss: 0.5590 -
val_loss: 0.4522
Epoch 12/100
413/413 [==============================] - 1s 4ms/step - loss: 0.5479 -
val_loss: 0.4492
Epoch 13/100
413/413 [==============================] - 1s 4ms/step - loss: 0.5364 -
val_loss: 0.4465
Epoch 14/100
413/413 [==============================] - 1s 3ms/step - loss: 0.5276 -
val_loss: 0.4642
Epoch 15/100
```

```
413/413 [==============================] - 1s 4ms/step - loss: 0.5217 -
val_loss: 0.4109
Epoch 16/100
413/413 [==============================] - 2s 4ms/step - loss: 0.5149 -
val_loss: 0.4267
Epoch 17/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4920 -
val_loss: 0.4211
Epoch 18/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4941 -
val_loss: 0.4476
Epoch 19/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4903 -
val_loss: 0.4552
Epoch 20/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4898 -
val_loss: 0.4043
Epoch 21/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4878 -
val_loss: 0.4226
Epoch 22/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4774 -
val_loss: 0.4364
Epoch 23/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4735 -
val_loss: 0.4281
Epoch 24/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4761 -
val_loss: 0.4301
Epoch 25/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4671 -
val_loss: 0.4387
Epoch 26/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4703 -
val_loss: 0.3719
Epoch 27/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4624 -
val_loss: 0.4174
Epoch 28/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4738 -
val_loss: 0.4359
Epoch 29/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4599 -
val_loss: 0.4242
Epoch 30/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4701 -
val_loss: 0.4113
Epoch 31/100
```

```
413/413 [==============================] - 1s 3ms/step - loss: 0.4475 -
val_loss: 0.4025
Epoch 32/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4590 -
val_loss: 0.4269
Epoch 33/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4478 -
val_loss: 0.4160
Epoch 34/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4558 -
val_loss: 0.4261
Epoch 35/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4547 -
val_loss: 0.4232
Epoch 36/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4555 -
val_loss: 0.3926
Epoch 37/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4452 -
val_loss: 0.3989
Epoch 38/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4468 -
val_loss: 0.3890
Epoch 39/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4475 -
val_loss: 0.4201
Epoch 40/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4503 -
val_loss: 0.4610
Epoch 41/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4451 -
val_loss: 0.4220
Epoch 42/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4423 -
val_loss: 0.3853
Epoch 43/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4443 -
val_loss: 0.3621
Epoch 44/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4400 -
val_loss: 0.3973
Epoch 45/100
413/413 [==============================] - 3s 7ms/step - loss: 0.4495 -
val_loss: 0.4068
Epoch 46/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4399 -
val_loss: 0.4026
Epoch 47/100
```

```
413/413 [==============================] - 1s 3ms/step - loss: 0.4439 -
val_loss: 0.4401
Epoch 48/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4338 -
val_loss: 0.4004
Epoch 49/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4350 -
val_loss: 0.3635
Epoch 50/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4320 -
val_loss: 0.4187
Epoch 51/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4350 -
val_loss: 0.3521
Epoch 52/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4280 -
val_loss: 0.3537
Epoch 53/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4332 -
val_loss: 0.3712
Epoch 54/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4374 -
val_loss: 0.4304
Epoch 55/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4390 -
val_loss: 0.3997
Epoch 56/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4310 -
val_loss: 0.3677
Epoch 57/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4227 -
val_loss: 0.3879
Epoch 58/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4357 -
val_loss: 0.3924
Epoch 59/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4239 -
val_loss: 0.3886
Epoch 60/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4315 -
val_loss: 0.4155
Epoch 61/100
413/413 [==============================] - 2s 6ms/step - loss: 0.4415 -
val_loss: 0.4076
Epoch 62/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4330 -
val_loss: 0.3750
Epoch 63/100
```

```
413/413 [==============================] - 2s 5ms/step - loss: 0.4303 -
val_loss: 0.4030
Epoch 64/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4299 -
val_loss: 0.4044
Epoch 65/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4343 -
val_loss: 0.3533
Epoch 66/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4242 -
val_loss: 0.3945
Epoch 67/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4318 -
val_loss: 0.3346
Epoch 68/100
413/413 [==============================] - 2s 6ms/step - loss: 0.4276 -
val_loss: 0.3703
Epoch 69/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4265 -
val_loss: 0.3738
Epoch 70/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4258 -
val_loss: 0.3632
Epoch 71/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4300 -
val_loss: 0.3561
Epoch 72/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4241 -
val_loss: 0.3641
Epoch 73/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4274 -
val_loss: 0.3880
Epoch 74/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4236 -
val_loss: 0.3790
Epoch 75/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4293 -
val_loss: 0.3749
Epoch 76/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4290 -
val_loss: 0.3405
Epoch 77/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4265 -
val_loss: 0.3862
Epoch 78/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4297 -
val_loss: 0.3862
Epoch 79/100
```

```
413/413 [==============================] - 2s 4ms/step - loss: 0.4246 -
val_loss: 0.3907
Epoch 80/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4293 -
val_loss: 0.3636
Epoch 81/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4214 -
val_loss: 0.3748
Epoch 82/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4233 -
val_loss: 0.3650
Epoch 83/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4190 -
val_loss: 0.3906
Epoch 84/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4215 -
val_loss: 0.3784
Epoch 85/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4164 -
val_loss: 0.3603
Epoch 86/100
413/413 [==============================] - 2s 5ms/step - loss: 0.4204 -
val_loss: 0.4106
Epoch 87/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4172 -
val_loss: 0.3766
Epoch 88/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4231 -
val_loss: 0.4074
Epoch 89/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4222 -
val_loss: 0.3841
Epoch 90/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4184 -
val_loss: 0.4094
Epoch 91/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4263 -
val_loss: 0.3810
Epoch 92/100
413/413 [==============================] - 1s 4ms/step - loss: 0.4225 -
val_loss: 0.3592
Epoch 93/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4298 -
val_loss: 0.3650
Epoch 94/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4243 -
val_loss: 0.3700
Epoch 95/100
```

```
413/413 [==============================] - 2s 5ms/step - loss: 0.4191 -
val_loss: 0.4143
Epoch 96/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4204 -
val_loss: 0.3958
Epoch 97/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4209 -
val_loss: 0.3722
Epoch 98/100
413/413 [==============================] - 1s 3ms/step - loss: 0.4178 -
val_loss: 0.3941
Epoch 99/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4216 -
val_loss: 0.3693
Epoch 100/100
413/413 [==============================] - 2s 4ms/step - loss: 0.4195 -
val_loss: 0.3804
```

[ ]:
```python
# Evaluar el modelo
loss = model.evaluate(x_test, y_test)
print(f'Pérdida en el conjunto de prueba: {loss}')
```

```
129/129 [==============================] - 0s 3ms/step - loss: 0.3900
Pérdida en el conjunto de prueba: 0.3900304138660431
```

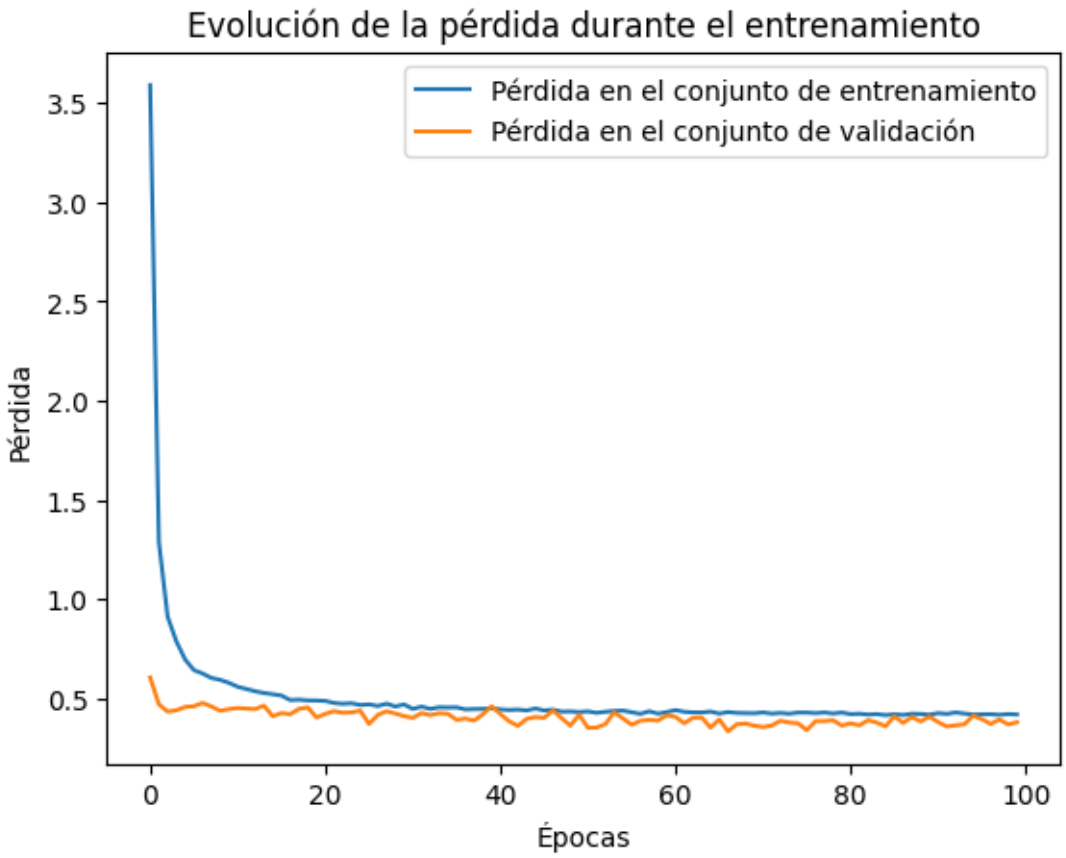
[ ]:
```python
# Hacer predicciones
y_pred = model.predict(x_test)
```

```
129/129 [==============================] - 1s 7ms/step
```

**Visualizar la Pérdida durante el Entrenamiento**

[ ]:
```python
import matplotlib.pyplot as plt

# Graficar la pérdida durante el entrenamiento
plt.plot(history.history['loss'], label='Pérdida en el conjunto de␣
 ↪entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en el conjunto de␣
 ↪validación')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
plt.title('Evolución de la pérdida durante el entrenamiento')
plt.show()
```




**Evolución de la Pérdida durante el Entrenamiento**

La gráfica muestra cómo la pérdida disminuye a medida que avanzan las épocas tanto en el conjunto de entrenamiento como en el de validación.

**Observaciones:** * La pérdida en el conjunto de entrenamiento (línea azul) disminuye rápidamente al principio y luego se estabiliza. * La pérdida en el conjunto de validación (línea naranja) también disminuye y se mantiene baja, lo que indica que el modelo está generalizando bien y no está sobreajustando los datos de entrenamiento.

[ ]:
```
# Evaluar el modelo en el conjunto de prueba
loss = model.evaluate(x_test, y_test)
print(f'Pérdida en el conjunto de prueba: {loss}')
```

```
129/129 [==============================] - 1s 7ms/step - loss: 0.3900
Pérdida en el conjunto de prueba: 0.3900304138660431
```

**Pérdida en el Conjunto de Prueba**

**Interpretación:** La pérdida (loss) en el conjunto de prueba es aproximadamente 0.39. Esta métrica indica el error cuadrático medio (mean squared error) entre las predicciones del modelo y los valores reales. Un valor más bajo de pérdida es mejor, ya que indica que las predicciones están

más cerca de los valores reales.

[ ]:
```
# Hacer predicciones en el conjunto de prueba
y_pred = model.predict(x_test)

# Comparar predicciones con valores reales
comparison_df = pd.DataFrame({'Valor Real': y_test, 'Predicción': y_pred.
 ↪flatten()})

# Mostrar las primeras 10 comparaciones
print(comparison_df.head(10))
```

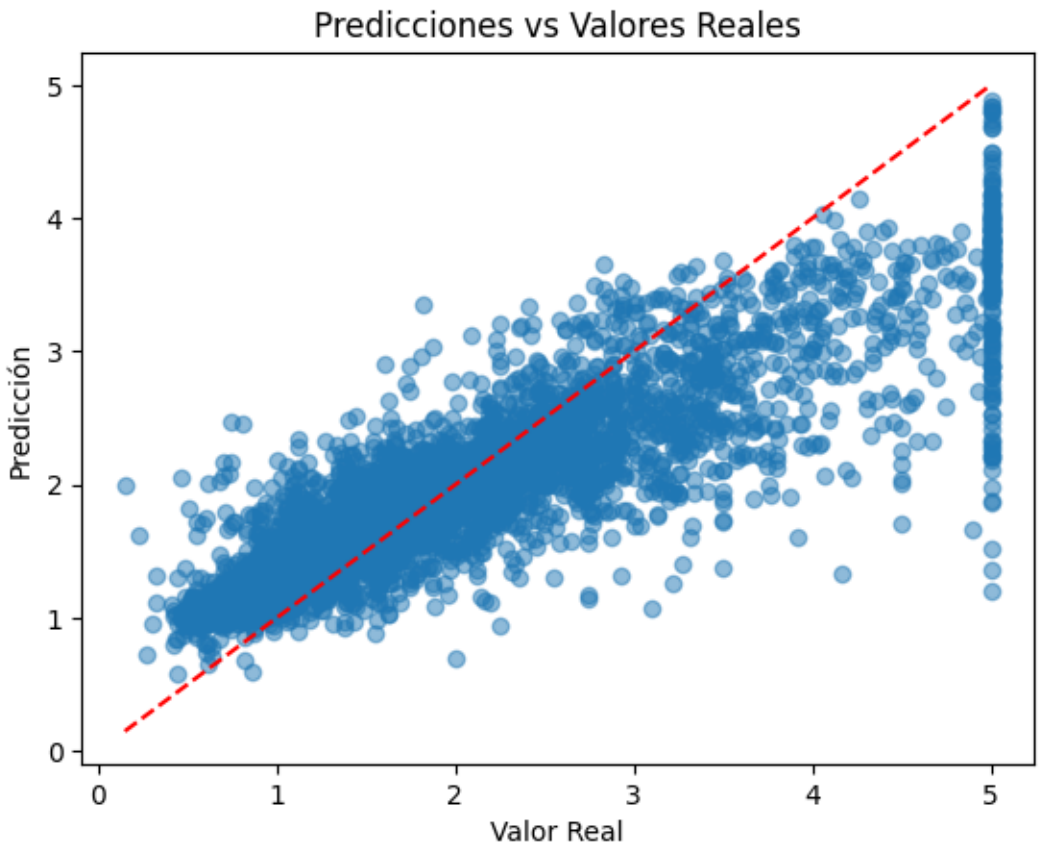
```
129/129 [==============================] - 1s 5ms/step
       Valor Real  Predicción
14740       1.369    1.553452
10101       2.413    2.388838
20566       2.007    1.566428
2670        0.725    0.991596
15709       4.600    3.270287
439         1.200    1.981063
845         2.470    2.407481
3768        3.369    3.293255
964         3.397    2.563825
8681        2.656    2.097074
```

**Comparación de Predicciones con Valores Reales**

**Interpretación:** * La tabla muestra una comparación entre los valores reales y las predicciones del modelo para las primeras 10 muestras en el conjunto de prueba. * Las predicciones están razonablemente cerca de los valores reales, lo que indica un buen desempeño del modelo.

[ ]:
```
# Graficar predicciones vs valores reales
plt.scatter(y_test, y_pred, alpha=0.5)
plt.xlabel('Valor Real')
plt.ylabel('Predicción')
plt.title('Predicciones vs Valores Reales')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'r--')

# Línea de identidad
plt.show()
```




**Gráfica de Predicciones vs Valores Reales**

La gráfica de muestra cómo se distribuyen las predicciones del modelo en comparación con los valores reales.

**Observaciones:** * La mayoría de los puntos están cerca de la línea roja (línea de identidad), lo que indica que las predicciones son bastante precisas. * Sin embargo, hay algunos puntos que se desvían significativamente de la línea de identidad, lo que indica errores en algunas predicciones.

### 3. Modelo de Perceptrón Multicapa (MultiLayer Perceptron, MLP) o Red Neuronal Profunda (Deep Neural Network, DNN)

Desarrollé un Modelo de Perceptrón Multicapa (MultiLayer Perceptron, MLP) para un problema de regresión utilizando Keras. Comenzamos inicializando un modelo secuencial y añadiendo una capa oculta con 16 neuronas y función de activación ReLU, seguida de una segunda capa oculta con 8 neuronas y la misma función de activación. La capa de salida consistió en una sola neurona con función de activación lineal, adecuada para predicciones de valores continuos. Compilamos el modelo utilizando el optimizador de descenso de gradiente estocástico (SGD) y la función de pérdida de error cuadrático medio (MSE). Implementamos Early Stopping para detener el entrenamiento automáticamente si la pérdida en el conjunto de validación no mejoraba después de 10 épocas consecutivas. El entrenamiento se detuvo en la época 232, indicando que el modelo alcanzó un punto de estabilidad. La evolución de la pérdida mostró una buena convergencia tanto en los

conjuntos de entrenamiento como en los de validación, lo que sugiere que el modelo generaliza bien y no está sobreajustando.

[ ]:
```
# Importar las bibliotecas necesarias
import tensorflow as tf
from tensorflow import keras
from keras import Sequential
from keras.layers import Dense, Dropout, BatchNormalization
from keras.callbacks import EarlyStopping
```

[ ]:
```
# Inicializar el modelo secuencial
dnn = Sequential()
```

[ ]:
```
# Añadir la primera capa oculta con 16 neuronas y función de activación ReLU
# input_dim=8 especifica que la entrada tiene 8 características
dnn.add(Dense(16, activation='relu', input_dim=8))
```

[ ]:
```
# (Opcional) Añadir normalización por lotes para estabilizar y acelerar el␣
↪entrenamiento
# dnn.add(BatchNormalization())

# (Opcional) Añadir Dropout para evitar el sobreajuste
# dnn.add(Dropout(0.25))

# Añadir una segunda capa oculta con 8 neuronas y función de activación ReLU
dnn.add(Dense(8, activation='relu'))
```

[ ]:
```
# (Opcional) Añadir normalización por lotes
# dnn.add(BatchNormalization())

# (Opcional) Añadir Dropout para evitar el sobreajuste
# dnn.add(Dropout(0.5))

# Añadir la capa de salida con 1 neurona y función de activación lineal
# Usamos una neurona porque estamos prediciendo un valor continuo
dnn.add(Dense(1, activation='linear'))
```

[ ]:
```
# Mostramos un resumen del modelo
dnn.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_6 (Dense)             (None, 16)                144

=================================================================
Total params: 144 (576.00 Byte)
```

```
Trainable params: 144 (576.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

[ ]:
```
# Compilamos el modelo usando la función de optimización y pérdida
dnn.compile(optimizer='sgd', loss='mean_squared_error')
```

[ ]:
```
# Configuramos Early Stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=10,␣
 ↪restore_best_weights=True)
```

**Beneficios de Early Stopping**

- Evita el Sobreajuste: Detiene el entrenamiento antes de que el modelo comience a sobreajustarse a los datos de entrenamiento.
- Ahorra Tiempo: Reduce el tiempo de entrenamiento al detenerse automáticamente cuando ya no se observa una mejora.

[ ]:
```
# Entrenar el modelo con los datos de entrenamiento
# validation_split=0.2 reserva el 20% de los datos de entrenamiento para␣
 ↪validación
history = dnn.fit(x_train, y_train, epochs=500, validation_split=0.2,␣
 ↪callbacks=[early_stopping])
```

```
Epoch 1/500
413/413 [==============================] - 2s 4ms/step - loss: 4.3349 -
val_loss: 3.7673
Epoch 2/500
413/413 [==============================] - 1s 3ms/step - loss: 3.3555 -
val_loss: 2.8564
Epoch 3/500
413/413 [==============================] - 1s 2ms/step - loss: 2.5000 -
val_loss: 2.1031
Epoch 4/500
413/413 [==============================] - 1s 2ms/step - loss: 1.8236 -
val_loss: 1.5111
Epoch 5/500
413/413 [==============================] - 1s 2ms/step - loss: 1.3065 -
val_loss: 1.0796
Epoch 6/500
413/413 [==============================] - 1s 2ms/step - loss: 0.9567 -
val_loss: 0.8166
Epoch 7/500
413/413 [==============================] - 1s 2ms/step - loss: 0.7604 -
val_loss: 0.6824
Epoch 8/500
413/413 [==============================] - 1s 2ms/step - loss: 0.6702 -
val_loss: 0.6247
```

```
Epoch 9/500
413/413 [==============================] - 1s 2ms/step - loss: 0.6325 -
val_loss: 0.5996
Epoch 10/500
413/413 [==============================] - 1s 2ms/step - loss: 0.6150 -
val_loss: 0.5865
Epoch 11/500
413/413 [==============================] - 1s 2ms/step - loss: 0.6047 -
val_loss: 0.5777
Epoch 12/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5973 -
val_loss: 0.5710
Epoch 13/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5912 -
val_loss: 0.5652
Epoch 14/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5859 -
val_loss: 0.5602
Epoch 15/500
413/413 [==============================] - 1s 4ms/step - loss: 0.5810 -
val_loss: 0.5554
Epoch 16/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5765 -
val_loss: 0.5508
Epoch 17/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5722 -
val_loss: 0.5467
Epoch 18/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5683 -
val_loss: 0.5427
Epoch 19/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5644 -
val_loss: 0.5387
Epoch 20/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5608 -
val_loss: 0.5352
Epoch 21/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5575 -
val_loss: 0.5321
Epoch 22/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5544 -
val_loss: 0.5290
Epoch 23/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5516 -
val_loss: 0.5263
Epoch 24/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5489 -
val_loss: 0.5236
```

```
Epoch 25/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5463 -
val_loss: 0.5210
Epoch 26/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5440 -
val_loss: 0.5188
Epoch 27/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5418 -
val_loss: 0.5166
Epoch 28/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5399 -
val_loss: 0.5145
Epoch 29/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5381 -
val_loss: 0.5131
Epoch 30/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5365 -
val_loss: 0.5112
Epoch 31/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5348 -
val_loss: 0.5096
Epoch 32/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5332 -
val_loss: 0.5078
Epoch 33/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5318 -
val_loss: 0.5065
Epoch 34/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5305 -
val_loss: 0.5053
Epoch 35/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5292 -
val_loss: 0.5044
Epoch 36/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5282 -
val_loss: 0.5028
Epoch 37/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5272 -
val_loss: 0.5015
Epoch 38/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5263 -
val_loss: 0.5006
Epoch 39/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5254 -
val_loss: 0.4995
Epoch 40/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5245 -
val_loss: 0.4990
```

```
Epoch 41/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5238 -
val_loss: 0.4980
Epoch 42/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5230 -
val_loss: 0.4977
Epoch 43/500
413/413 [==============================] - 2s 4ms/step - loss: 0.5223 -
val_loss: 0.4968
Epoch 44/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5216 -
val_loss: 0.4961
Epoch 45/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5209 -
val_loss: 0.4949
Epoch 46/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5204 -
val_loss: 0.4941
Epoch 47/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5198 -
val_loss: 0.4932
Epoch 48/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5193 -
val_loss: 0.4926
Epoch 49/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5188 -
val_loss: 0.4922
Epoch 50/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5183 -
val_loss: 0.4921
Epoch 51/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5178 -
val_loss: 0.4917
Epoch 52/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5173 -
val_loss: 0.4910
Epoch 53/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5169 -
val_loss: 0.4903
Epoch 54/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5165 -
val_loss: 0.4898
Epoch 55/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5161 -
val_loss: 0.4896
Epoch 56/500
413/413 [==============================] - 1s 4ms/step - loss: 0.5157 -
val_loss: 0.4894
```

```
Epoch 57/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5153 -
val_loss: 0.4892
Epoch 58/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5150 -
val_loss: 0.4888
Epoch 59/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5146 -
val_loss: 0.4886
Epoch 60/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5143 -
val_loss: 0.4882
Epoch 61/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5138 -
val_loss: 0.4873
Epoch 62/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5135 -
val_loss: 0.4878
Epoch 63/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5133 -
val_loss: 0.4869
Epoch 64/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5130 -
val_loss: 0.4864
Epoch 65/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5127 -
val_loss: 0.4864
Epoch 66/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5125 -
val_loss: 0.4858
Epoch 67/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5122 -
val_loss: 0.4854
Epoch 68/500
413/413 [==============================] - 2s 4ms/step - loss: 0.5120 -
val_loss: 0.4853
Epoch 69/500
413/413 [==============================] - 1s 4ms/step - loss: 0.5116 -
val_loss: 0.4846
Epoch 70/500
413/413 [==============================] - 2s 4ms/step - loss: 0.5114 -
val_loss: 0.4841
Epoch 71/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5112 -
val_loss: 0.4849
Epoch 72/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5108 -
val_loss: 0.4841
```

```
Epoch 73/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5107 -
val_loss: 0.4838
Epoch 74/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5104 -
val_loss: 0.4834
Epoch 75/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5102 -
val_loss: 0.4838
Epoch 76/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5100 -
val_loss: 0.4838
Epoch 77/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5098 -
val_loss: 0.4835
Epoch 78/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5095 -
val_loss: 0.4838
Epoch 79/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5094 -
val_loss: 0.4836
Epoch 80/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5091 -
val_loss: 0.4834
Epoch 81/500
413/413 [==============================] - 1s 4ms/step - loss: 0.5087 -
val_loss: 0.4825
Epoch 82/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5086 -
val_loss: 0.4836
Epoch 83/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5085 -
val_loss: 0.4833
Epoch 84/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5080 -
val_loss: 0.4816
Epoch 85/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5081 -
val_loss: 0.4818
Epoch 86/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5078 -
val_loss: 0.4813
Epoch 87/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5077 -
val_loss: 0.4815
Epoch 88/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5074 -
val_loss: 0.4809
```

```
Epoch 89/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5073 -
val_loss: 0.4810
Epoch 90/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5069 -
val_loss: 0.4803
Epoch 91/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5070 -
val_loss: 0.4805
Epoch 92/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5067 -
val_loss: 0.4803
Epoch 93/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5065 -
val_loss: 0.4801
Epoch 94/500
413/413 [==============================] - 2s 5ms/step - loss: 0.5065 -
val_loss: 0.4803
Epoch 95/500
413/413 [==============================] - 2s 5ms/step - loss: 0.5061 -
val_loss: 0.4795
Epoch 96/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5060 -
val_loss: 0.4798
Epoch 97/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5059 -
val_loss: 0.4801
Epoch 98/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5055 -
val_loss: 0.4790
Epoch 99/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5055 -
val_loss: 0.4796
Epoch 100/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5053 -
val_loss: 0.4790
Epoch 101/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5050 -
val_loss: 0.4784
Epoch 102/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5050 -
val_loss: 0.4783
Epoch 103/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5051 -
val_loss: 0.4781
Epoch 104/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5049 -
val_loss: 0.4782
```

```
Epoch 105/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5045 -
val_loss: 0.4790
Epoch 106/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5043 -
val_loss: 0.4789
Epoch 107/500
413/413 [==============================] - 1s 4ms/step - loss: 0.5042 -
val_loss: 0.4783
Epoch 108/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5042 -
val_loss: 0.4784
Epoch 109/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5037 -
val_loss: 0.4774
Epoch 110/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5038 -
val_loss: 0.4771
Epoch 111/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5035 -
val_loss: 0.4767
Epoch 112/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5036 -
val_loss: 0.4765
Epoch 113/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5035 -
val_loss: 0.4765
Epoch 114/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5034 -
val_loss: 0.4769
Epoch 115/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5031 -
val_loss: 0.4770
Epoch 116/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5027 -
val_loss: 0.4762
Epoch 117/500
413/413 [==============================] - 2s 4ms/step - loss: 0.5028 -
val_loss: 0.4758
Epoch 118/500
413/413 [==============================] - 2s 6ms/step - loss: 0.5028 -
val_loss: 0.4762
Epoch 119/500
413/413 [==============================] - 2s 4ms/step - loss: 0.5025 -
val_loss: 0.4765
Epoch 120/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5023 -
val_loss: 0.4761
```

```
Epoch 121/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5021 -
val_loss: 0.4758
Epoch 122/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5020 -
val_loss: 0.4759
Epoch 123/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5018 -
val_loss: 0.4752
Epoch 124/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5020 -
val_loss: 0.4751
Epoch 125/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5018 -
val_loss: 0.4751
Epoch 126/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5016 -
val_loss: 0.4751
Epoch 127/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5016 -
val_loss: 0.4748
Epoch 128/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5012 -
val_loss: 0.4745
Epoch 129/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5011 -
val_loss: 0.4747
Epoch 130/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5008 -
val_loss: 0.4743
Epoch 131/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5008 -
val_loss: 0.4746
Epoch 132/500
413/413 [==============================] - 1s 4ms/step - loss: 0.5006 -
val_loss: 0.4740
Epoch 133/500
413/413 [==============================] - 1s 3ms/step - loss: 0.5005 -
val_loss: 0.4738
Epoch 134/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5006 -
val_loss: 0.4739
Epoch 135/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5001 -
val_loss: 0.4732
Epoch 136/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5006 -
val_loss: 0.4731
```

```
Epoch 137/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5004 -
val_loss: 0.4731
Epoch 138/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5003 -
val_loss: 0.4730
Epoch 139/500
413/413 [==============================] - 1s 2ms/step - loss: 0.5002 -
val_loss: 0.4729
Epoch 140/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4997 -
val_loss: 0.4736
Epoch 141/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4995 -
val_loss: 0.4737
Epoch 142/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4994 -
val_loss: 0.4737
Epoch 143/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4992 -
val_loss: 0.4732
Epoch 144/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4990 -
val_loss: 0.4725
Epoch 145/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4991 -
val_loss: 0.4730
Epoch 146/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4989 -
val_loss: 0.4729
Epoch 147/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4994 -
val_loss: 0.4725
Epoch 148/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4987 -
val_loss: 0.4723
Epoch 149/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4985 -
val_loss: 0.4718
Epoch 150/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4984 -
val_loss: 0.4726
Epoch 151/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4982 -
val_loss: 0.4719
Epoch 152/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4983 -
val_loss: 0.4723
```

```
Epoch 153/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4979 -
val_loss: 0.4717
Epoch 154/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4979 -
val_loss: 0.4712
Epoch 155/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4978 -
val_loss: 0.4714
Epoch 156/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4977 -
val_loss: 0.4719
Epoch 157/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4974 -
val_loss: 0.4711
Epoch 158/500
413/413 [==============================] - 1s 4ms/step - loss: 0.4974 -
val_loss: 0.4706
Epoch 159/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4973 -
val_loss: 0.4704
Epoch 160/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4974 -
val_loss: 0.4708
Epoch 161/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4972 -
val_loss: 0.4709
Epoch 162/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4970 -
val_loss: 0.4715
Epoch 163/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4971 -
val_loss: 0.4711
Epoch 164/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4968 -
val_loss: 0.4711
Epoch 165/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4966 -
val_loss: 0.4710
Epoch 166/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4966 -
val_loss: 0.4712
Epoch 167/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4965 -
val_loss: 0.4714
Epoch 168/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4963 -
val_loss: 0.4707
```

```
Epoch 169/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4961 -
val_loss: 0.4700
Epoch 170/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4961 -
val_loss: 0.4698
Epoch 171/500
413/413 [==============================] - 2s 4ms/step - loss: 0.4961 -
val_loss: 0.4699
Epoch 172/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4958 -
val_loss: 0.4694
Epoch 173/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4957 -
val_loss: 0.4690
Epoch 174/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4956 -
val_loss: 0.4686
Epoch 175/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4956 -
val_loss: 0.4689
Epoch 176/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4955 -
val_loss: 0.4693
Epoch 177/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4960 -
val_loss: 0.4686
Epoch 178/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4956 -
val_loss: 0.4686
Epoch 179/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4951 -
val_loss: 0.4685
Epoch 180/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4951 -
val_loss: 0.4689
Epoch 181/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4954 -
val_loss: 0.4684
Epoch 182/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4952 -
val_loss: 0.4684
Epoch 183/500
413/413 [==============================] - 1s 4ms/step - loss: 0.4947 -
val_loss: 0.4682
Epoch 184/500
413/413 [==============================] - 2s 4ms/step - loss: 0.4946 -
val_loss: 0.4690
```

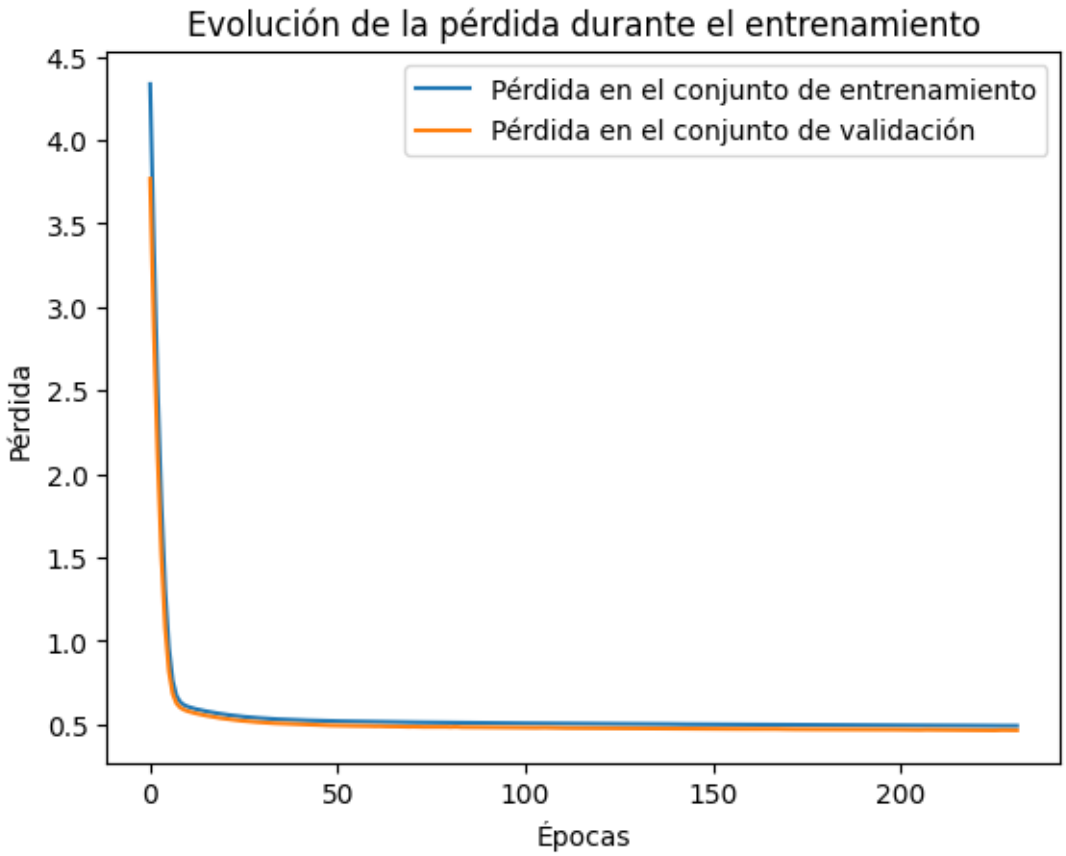
```
Epoch 185/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4944 -
val_loss: 0.4683
Epoch 186/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4949 -
val_loss: 0.4679
Epoch 187/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4946 -
val_loss: 0.4678
Epoch 188/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4942 -
val_loss: 0.4675
Epoch 189/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4945 -
val_loss: 0.4671
Epoch 190/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4940 -
val_loss: 0.4669
Epoch 191/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4943 -
val_loss: 0.4671
Epoch 192/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4938 -
val_loss: 0.4668
Epoch 193/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4937 -
val_loss: 0.4667
Epoch 194/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4939 -
val_loss: 0.4670
Epoch 195/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4935 -
val_loss: 0.4673
Epoch 196/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4933 -
val_loss: 0.4671
Epoch 197/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4932 -
val_loss: 0.4666
Epoch 198/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4932 -
val_loss: 0.4671
Epoch 199/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4931 -
val_loss: 0.4675
Epoch 200/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4929 -
val_loss: 0.4670
```

```
Epoch 201/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4928 -
val_loss: 0.4667
Epoch 202/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4926 -
val_loss: 0.4661
Epoch 203/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4927 -
val_loss: 0.4661
Epoch 204/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4928 -
val_loss: 0.4659
Epoch 205/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4925 -
val_loss: 0.4652
Epoch 206/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4925 -
val_loss: 0.4655
Epoch 207/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4923 -
val_loss: 0.4661
Epoch 208/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4922 -
val_loss: 0.4664
Epoch 209/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4919 -
val_loss: 0.4657
Epoch 210/500
413/413 [==============================] - 1s 4ms/step - loss: 0.4919 -
val_loss: 0.4662
Epoch 211/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4920 -
val_loss: 0.4656
Epoch 212/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4921 -
val_loss: 0.4654
Epoch 213/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4916 -
val_loss: 0.4655
Epoch 214/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4918 -
val_loss: 0.4653
Epoch 215/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4912 -
val_loss: 0.4643
Epoch 216/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4913 -
val_loss: 0.4643
```

```
Epoch 217/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4911 -
val_loss: 0.4638
Epoch 218/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4913 -
val_loss: 0.4639
Epoch 219/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4912 -
val_loss: 0.4639
Epoch 220/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4910 -
val_loss: 0.4643
Epoch 221/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4911 -
val_loss: 0.4640
Epoch 222/500
413/413 [==============================] - 2s 4ms/step - loss: 0.4904 -
val_loss: 0.4626
Epoch 223/500
413/413 [==============================] - 2s 4ms/step - loss: 0.4908 -
val_loss: 0.4631
Epoch 224/500
413/413 [==============================] - 2s 4ms/step - loss: 0.4906 -
val_loss: 0.4633
Epoch 225/500
413/413 [==============================] - 2s 4ms/step - loss: 0.4904 -
val_loss: 0.4632
Epoch 226/500
413/413 [==============================] - 1s 3ms/step - loss: 0.4902 -
val_loss: 0.4629
Epoch 227/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4900 -
val_loss: 0.4642
Epoch 228/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4900 -
val_loss: 0.4643
Epoch 229/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4903 -
val_loss: 0.4639
Epoch 230/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4898 -
val_loss: 0.4634
Epoch 231/500
413/413 [==============================] - 1s 2ms/step - loss: 0.4901 -
val_loss: 0.4633
Epoch 232/500
413/413 [==============================] - 2s 4ms/step - loss: 0.4898 -
val_loss: 0.4633
```

```
[ ]: import matplotlib.pyplot as plt

# Graficar la pérdida durante el entrenamiento
plt.plot(history.history['loss'], label='Pérdida en el conjunto de␣
 ↪entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en el conjunto de␣
 ↪validación')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
plt.title('Evolución de la pérdida durante el entrenamiento')
plt.show()
```




**Interpretación de la Gráfica de Pérdida**

1. Eje X (Épocas):

- Representa el número de épocas (iteraciones) durante el entrenamiento del modelo.
- El entrenamiento se detuvo en la época 232 debido a la configuración de Early Stopping.

2. Eje Y (Pérdida):

- Representa la función de pérdida (error cuadrático medio, MSE) calculada en cada época.
- Valores más bajos indican un mejor ajuste del modelo a los datos.

3. Línea Azul (Pérdida en el Conjunto de Entrenamiento):

- Muestra la evolución de la pérdida en el conjunto de entrenamiento.
- Inicialmente, la pérdida es alta, pero disminuye rápidamente y luego se estabiliza.

4. Línea Naranja (Pérdida en el Conjunto de Validación):

- Muestra la evolución de la pérdida en el conjunto de validación.
- Sigue una tendencia similar a la línea azul, lo que indica que el modelo está generalizando bien y no se está sobreajustando.

*Observaciones Clave:*

1. Disminución Rápida Inicial:

- Ambas líneas muestran una disminución rápida en la pérdida al inicio del entrenamiento. Esto es típico cuando el modelo comienza a aprender patrones significativos en los datos.

2. Estabilización de la Pérdida:

- Después de la rápida disminución inicial, ambas líneas se estabilizan alrededor de un valor de pérdida de aproximadamente 0.5.
- La estabilización indica que el modelo ha alcanzado un punto donde no se observan mejoras significativas adicionales con más iteraciones.

3. Concordancia entre Entrenamiento y Validación:

- La línea de pérdida en el conjunto de entrenamiento es muy similar a la línea de pérdida en el conjunto de validación.
- Esta concordancia sugiere que el modelo no está sobreajustando y tiene un buen rendimiento en datos no vistos.

4. Detención del Entrenamiento:

- El entrenamiento se detuvo en la época 232 debido a la configuración de Early Stopping.
- Esto es beneficioso ya que evita entrenar el modelo innecesariamente cuando ya no se observan mejoras.

**Conclusión:**

La gráfica muestra que el modelo se ha entrenado de manera efectiva:

- Eficiencia en el Entrenamiento: El modelo aprendió rápidamente en las primeras épocas y luego se estabilizó, lo que indica un buen ajuste.
- Generalización: La concordancia entre las pérdidas de entrenamiento y validación sugiere que el modelo generaliza bien y no se está sobreajustando.
- Uso de Early Stopping: La detención automática en la época 232 muestra que el uso de Early Stopping fue efectivo para evitar un entrenamiento innecesario y potencialmente perjudicial.

En resumen, el modelo entrenado parece tener un buen rendimiento tanto en el conjunto de entrenamiento como en el de validación, y la configuración de Early Stopping ayudó a optimizar el proceso de entrenamiento.

### 4. Problema de Clasificación

Abordamos un problema de clasificación utilizando el conjunto de datos del Titanic para predecir la supervivencia de los pasajeros. Primero, cargamos y exploramos los datos, imputando valores faltantes y seleccionando las características relevantes. Luego, convertimos las variables categóricas en variables dummy mediante one-hot encoding. Dividimos los datos en características (X) y etiquetas (y), y aplicamos la técnica de sobremuestreo SMOTE para balancear las clases. Posteriormente, escalamos las características y dividimos los datos en conjuntos de entrenamiento y prueba. Construimos y entrenamos un modelo de red neuronal utilizando Keras, evaluando su rendimiento con métricas como la pérdida y la precisión. Visualizamos la evolución de la pérdida y la precisión durante el entrenamiento. Finalmente, evaluamos el modelo en el conjunto de prueba, obteniendo una precisión del 82.73% y analizamos detalladamente las métricas de clasificación, que mostraron un buen equilibrio entre precisión y recall para ambas clases.

#### Descripción del Problema

El objetivo es predecir si un pasajero sobrevivió o no al desastre del Titanic basándose en características como su clase de pasajero, sexo, edad, número de hermanos/esposos a bordo, número de padres/hijos a bordo, tarifa pagada y lugar de embarque.

#### Cargar y Explorar los Datos

[ ]:
```
import seaborn as sns
import pandas as pd

# Cargar el conjunto de datos de Titanic
dataset = sns.load_dataset("titanic")

# Mostrar las primeras filas del dataset
dataset.head()
```

[ ]:
```
   survived  pclass     sex   age  sibsp  parch     fare embarked  class  \
0         0       3    male  22.0      1      0   7.2500        S  Third
1         1       1  female  38.0      1      0  71.2833        C  First
2         1       3  female  26.0      0      0   7.9250        S  Third
3         1       1  female  35.0      1      0  53.1000        S  First
4         0       3    male  35.0      0      0   8.0500        S  Third

     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
1  woman       False    C    Cherbourg   yes  False
2  woman       False  NaN  Southampton   yes   True
3  woman       False    C  Southampton   yes  False
4    man        True  NaN  Southampton    no   True
```

#### Verificar Valores Nulos y Tipos de Datos

[ ]:
```
# Verificar si hay valores nulos en el dataset
dataset.isnull().sum()
```

```
# Obtener información sobre el dataset
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    category
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object
 13  alive        891 non-null    object
 14  alone        891 non-null    bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
```

**Imputar Valores Faltantes y Seleccionar Columnas Relevantes**

[ ]:
```
# Imputar valores faltantes
dataset['age'].fillna(dataset['age'].mean(), inplace=True)
dataset['embarked'].fillna(dataset['embarked'].mode()[0], inplace=True)
```

[ ]:
```
# Eliminar la columna 'deck' ya que tiene muchos valores faltantes
dataset.drop(columns=['deck'], inplace=True)
```

[ ]:
```
# Eliminar filas con valores faltantes restantes
dataset.dropna(inplace=True)
```

[ ]:
```
# Seleccionar las columnas relevantes
dataset = dataset[['survived', 'pclass', 'sex', 'age', 'sibsp', 'parch',␣
 ↪'fare', 'embarked']]
```

[ ]:
```
# Mostrar las primeras filas del dataset actualizado
dataset.head()
```

[ ]:

| | survived | pclass | sex | age | sibsp | parch | fare | embarked |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 3 | male | 22.0 | 1 | 0 | 7.2500 | S |
| 1 | 1 | 1 | female | 38.0 | 1 | 0 | 71.2833 | C |

```
2         1       3  female  26.0      0      0   7.9250        S
3         1       1  female  35.0      1      0  53.1000        S
4         0       3    male  35.0      0      0   8.0500        S
```

### Codificar Variables Categóricas

[ ]:
```
# Convertir variables categóricas en variables dummy (one-hot encoding)
dataset = pd.get_dummies(dataset, columns=['sex', 'embarked'], drop_first=True)
```

[ ]:
```
# Mostrar las primeras filas del dataset actualizado
dataset.head()
```

[ ]:
```
   survived  pclass   age  sibsp  parch     fare  sex_male  embarked_Q  \
0         0       3  22.0      1      0   7.2500      True       False
1         1       1  38.0      1      0  71.2833     False       False
2         1       3  26.0      0      0   7.9250     False       False
3         1       1  35.0      1      0  53.1000     False       False
4         0       3  35.0      0      0   8.0500      True       False

   embarked_S
0        True
1       False
2        True
3        True
4        True
```

### Dividir el Conjunto de Datos en Características y Etiquetas

[ ]:
```
# Dividir el dataset en variables independientes (X) y la variable dependiente
↪(y)
X = dataset.drop(['survived'], axis=1)
y = dataset[['survived']]
```

[ ]:
```
# Mostrar las primeras filas de X e y
X.head()
y.head()
```

[ ]:
```
   survived
0         0
1         1
2         1
3         1
4         0
```

### Balancear los Datos

[ ]:
```
from imblearn.over_sampling import SMOTE
```

[ ]:
```
# Aplicar SMOTE para balancear las clases
smote = SMOTE()
X_smote, y_smote = smote.fit_resample(X, y)
```

[ ]:
```
# Mostramos la distribución de clases antes y después del balanceo
print(y.value_counts())
print(y_smote.value_counts())
```

```
survived
0           549
1           340
Name: count, dtype: int64
survived
0           549
1           549
Name: count, dtype: int64
```

### Dividir el Conjunto de Datos en Entrenamiento y Prueba

[ ]:
```
from sklearn.model_selection import train_test_split
```

[ ]:
```
# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_smote, y_smote,␣
↪test_size=0.2, random_state=0)
```

### Escalar las Características

[ ]:
```
from sklearn.preprocessing import StandardScaler
```

[ ]:
```
# Escalar las características
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

### Entrenamiento

[ ]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

[ ]:
```
# Construccion del modelo
model = Sequential()
model.add(Dense(16, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

[ ]:
```
# Compilar el modelo
```

```
model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

[ ]:
```
# Entrenar el modelo
history = model.fit(X_train, y_train, epochs=50, batch_size=32,␣
 ↪validation_data=(X_test, y_test))
```

```
Epoch 1/50
28/28 [==============================] - 3s 33ms/step - loss: 0.6893 - accuracy:
0.6002 - val_loss: 0.6679 - val_accuracy: 0.6182
Epoch 2/50
28/28 [==============================] - 0s 12ms/step - loss: 0.6271 - accuracy:
0.6731 - val_loss: 0.6253 - val_accuracy: 0.6773
Epoch 3/50
28/28 [==============================] - 0s 11ms/step - loss: 0.5863 - accuracy:
0.7175 - val_loss: 0.5916 - val_accuracy: 0.7000
Epoch 4/50
28/28 [==============================] - 0s 9ms/step - loss: 0.5545 - accuracy:
0.7403 - val_loss: 0.5658 - val_accuracy: 0.7227
Epoch 5/50
28/28 [==============================] - 0s 5ms/step - loss: 0.5317 - accuracy:
0.7551 - val_loss: 0.5478 - val_accuracy: 0.7227
Epoch 6/50
28/28 [==============================] - 0s 7ms/step - loss: 0.5147 - accuracy:
0.7574 - val_loss: 0.5333 - val_accuracy: 0.7273
Epoch 7/50
28/28 [==============================] - 0s 8ms/step - loss: 0.5022 - accuracy:
0.7597 - val_loss: 0.5228 - val_accuracy: 0.7318
Epoch 8/50
28/28 [==============================] - 0s 6ms/step - loss: 0.4940 - accuracy:
0.7654 - val_loss: 0.5136 - val_accuracy: 0.7364
Epoch 9/50
28/28 [==============================] - 0s 7ms/step - loss: 0.4870 - accuracy:
0.7665 - val_loss: 0.5066 - val_accuracy: 0.7455
Epoch 10/50
28/28 [==============================] - 0s 5ms/step - loss: 0.4814 - accuracy:
0.7677 - val_loss: 0.5014 - val_accuracy: 0.7500
Epoch 11/50
28/28 [==============================] - 0s 6ms/step - loss: 0.4774 - accuracy:
0.7699 - val_loss: 0.4964 - val_accuracy: 0.7545
Epoch 12/50
28/28 [==============================] - 0s 7ms/step - loss: 0.4737 - accuracy:
0.7711 - val_loss: 0.4912 - val_accuracy: 0.7591
Epoch 13/50
28/28 [==============================] - 0s 8ms/step - loss: 0.4714 - accuracy:
0.7745 - val_loss: 0.4895 - val_accuracy: 0.7682
Epoch 14/50
```

```
28/28 [==============================] - 0s 8ms/step - loss: 0.4681 - accuracy:
0.7927 - val_loss: 0.4854 - val_accuracy: 0.7636
Epoch 15/50
28/28 [==============================] - 0s 7ms/step - loss: 0.4656 - accuracy:
0.7984 - val_loss: 0.4827 - val_accuracy: 0.7636
Epoch 16/50
28/28 [==============================] - 0s 9ms/step - loss: 0.4634 - accuracy:
0.7984 - val_loss: 0.4804 - val_accuracy: 0.7591
Epoch 17/50
28/28 [==============================] - 0s 6ms/step - loss: 0.4611 - accuracy:
0.8041 - val_loss: 0.4770 - val_accuracy: 0.7773
Epoch 18/50
28/28 [==============================] - 0s 7ms/step - loss: 0.4596 - accuracy:
0.8030 - val_loss: 0.4749 - val_accuracy: 0.7773
Epoch 19/50
28/28 [==============================] - 0s 7ms/step - loss: 0.4575 - accuracy:
0.8064 - val_loss: 0.4734 - val_accuracy: 0.7727
Epoch 20/50
28/28 [==============================] - 0s 8ms/step - loss: 0.4556 - accuracy:
0.8030 - val_loss: 0.4713 - val_accuracy: 0.7727
Epoch 21/50
28/28 [==============================] - 0s 9ms/step - loss: 0.4537 - accuracy:
0.8018 - val_loss: 0.4672 - val_accuracy: 0.7727
Epoch 22/50
28/28 [==============================] - 0s 6ms/step - loss: 0.4523 - accuracy:
0.8064 - val_loss: 0.4657 - val_accuracy: 0.7773
Epoch 23/50
28/28 [==============================] - 0s 7ms/step - loss: 0.4506 - accuracy:
0.8098 - val_loss: 0.4645 - val_accuracy: 0.7818
Epoch 24/50
28/28 [==============================] - 0s 7ms/step - loss: 0.4492 - accuracy:
0.8109 - val_loss: 0.4630 - val_accuracy: 0.8091
Epoch 25/50
28/28 [==============================] - 0s 7ms/step - loss: 0.4473 - accuracy:
0.8155 - val_loss: 0.4610 - val_accuracy: 0.8091
Epoch 26/50
28/28 [==============================] - 0s 8ms/step - loss: 0.4463 - accuracy:
0.8132 - val_loss: 0.4593 - val_accuracy: 0.8045
Epoch 27/50
28/28 [==============================] - 0s 6ms/step - loss: 0.4446 - accuracy:
0.8121 - val_loss: 0.4592 - val_accuracy: 0.8045
Epoch 28/50
28/28 [==============================] - 0s 5ms/step - loss: 0.4429 - accuracy:
0.8155 - val_loss: 0.4576 - val_accuracy: 0.8045
Epoch 29/50
28/28 [==============================] - 0s 3ms/step - loss: 0.4414 - accuracy:
0.8155 - val_loss: 0.4567 - val_accuracy: 0.8045
Epoch 30/50
```

```
28/28 [==============================] - 0s 4ms/step - loss: 0.4398 - accuracy:
0.8166 - val_loss: 0.4550 - val_accuracy: 0.8136
Epoch 31/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4391 - accuracy:
0.8166 - val_loss: 0.4519 - val_accuracy: 0.8091
Epoch 32/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4370 - accuracy:
0.8166 - val_loss: 0.4524 - val_accuracy: 0.8136
Epoch 33/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4356 - accuracy:
0.8155 - val_loss: 0.4496 - val_accuracy: 0.8182
Epoch 34/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4345 - accuracy:
0.8178 - val_loss: 0.4477 - val_accuracy: 0.8182
Epoch 35/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4327 - accuracy:
0.8178 - val_loss: 0.4450 - val_accuracy: 0.8182
Epoch 36/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4307 - accuracy:
0.8223 - val_loss: 0.4421 - val_accuracy: 0.8182
Epoch 37/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4301 - accuracy:
0.8166 - val_loss: 0.4403 - val_accuracy: 0.8136
Epoch 38/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4273 - accuracy:
0.8166 - val_loss: 0.4386 - val_accuracy: 0.8182
Epoch 39/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4267 - accuracy:
0.8189 - val_loss: 0.4380 - val_accuracy: 0.8182
Epoch 40/50
28/28 [==============================] - 0s 6ms/step - loss: 0.4265 - accuracy:
0.8178 - val_loss: 0.4350 - val_accuracy: 0.8136
Epoch 41/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4233 - accuracy:
0.8178 - val_loss: 0.4373 - val_accuracy: 0.8182
Epoch 42/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4227 - accuracy:
0.8200 - val_loss: 0.4350 - val_accuracy: 0.8227
Epoch 43/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4210 - accuracy:
0.8166 - val_loss: 0.4348 - val_accuracy: 0.8227
Epoch 44/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4200 - accuracy:
0.8178 - val_loss: 0.4350 - val_accuracy: 0.8182
Epoch 45/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4196 - accuracy:
0.8200 - val_loss: 0.4335 - val_accuracy: 0.8227
Epoch 46/50
```

```
28/28 [==============================] - 0s 3ms/step - loss: 0.4197 - accuracy:
0.8144 - val_loss: 0.4333 - val_accuracy: 0.8227
Epoch 47/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4185 - accuracy:
0.8200 - val_loss: 0.4338 - val_accuracy: 0.8182
Epoch 48/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4175 - accuracy:
0.8200 - val_loss: 0.4323 - val_accuracy: 0.8182
Epoch 49/50
28/28 [==============================] - 0s 5ms/step - loss: 0.4175 - accuracy:
0.8200 - val_loss: 0.4334 - val_accuracy: 0.8227
Epoch 50/50
28/28 [==============================] - 0s 4ms/step - loss: 0.4162 - accuracy:
0.8200 - val_loss: 0.4303 - val_accuracy: 0.8273
```

**Evaluación del modelo**

[ ]:
```python
# Evaluar el modelo en el conjunto de prueba
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Pérdida en el conjunto de prueba: {loss}')
print(f'Precisión en el conjunto de prueba: {accuracy}')
```

```
7/7 [==============================] - 0s 4ms/step - loss: 0.4303 - accuracy:
0.8273
Pérdida en el conjunto de prueba: 0.4303216338157654
Precisión en el conjunto de prueba: 0.8272727131843567
```

**Interpretación de los Resultados**

Al evaluar el modelo en el conjunto de prueba, obtuvimos una pérdida de 0.4303 y una precisión de 0.8273. Estos resultados indican que el modelo tiene un rendimiento bastante bueno en la tarea de predecir la supervivencia de los pasajeros del Titanic.

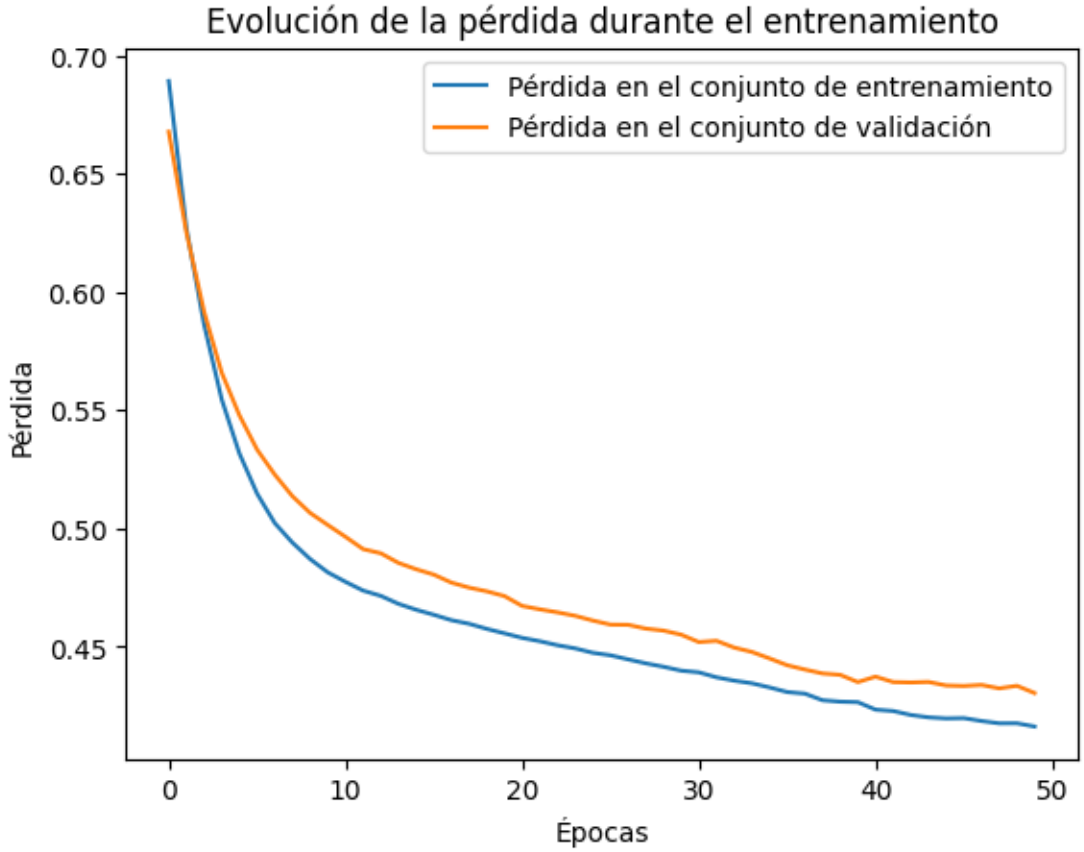
- Pérdida en el Conjunto de Prueba: 0.4303
- La pérdida representa el error del modelo en el conjunto de prueba. En este caso, una pérdida de 0.4303 es relativamente baja, lo que sugiere que el modelo está haciendo buenas predicciones. La función de pérdida utilizada es el binary_crossentropy, que es adecuada para problemas de clasificación binaria como este.
- Precisión en el Conjunto de Prueba: 0.8273
- La precisión mide el porcentaje de predicciones correctas hechas por el modelo en el conjunto de prueba. Con una precisión de 82.73%, podemos concluir que el modelo clasifica correctamente la supervivencia de los pasajeros en más del 82% de los casos. Esto indica que el modelo generaliza bien y tiene un buen rendimiento predictivo.

**Visualizar la evolución del entrenamiento**

[ ]:
```python
import matplotlib.pyplot as plt
```

[ ]:
```python
# Graficar la pérdida durante el entrenamiento
```

```
plt.plot(history.history['loss'], label='Pérdida en el conjunto de␣
 ↪entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en el conjunto de␣
 ↪validación')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
plt.title('Evolución de la pérdida durante el entrenamiento')
plt.show()
```




1. Eje X (Épocas):

- Representa el número de épocas (iteraciones) durante el entrenamiento del modelo.

2. Eje Y (Pérdida):

- Representa el valor de la función de pérdida (error) calculada en cada época.

3. Línea Azul (Pérdida en el Conjunto de Entrenamiento):

- Muestra la evolución de la pérdida en el conjunto de entrenamiento.

- La pérdida disminuye rápidamente al inicio, lo que indica que el modelo está aprendiendo bien los patrones de los datos de entrenamiento.

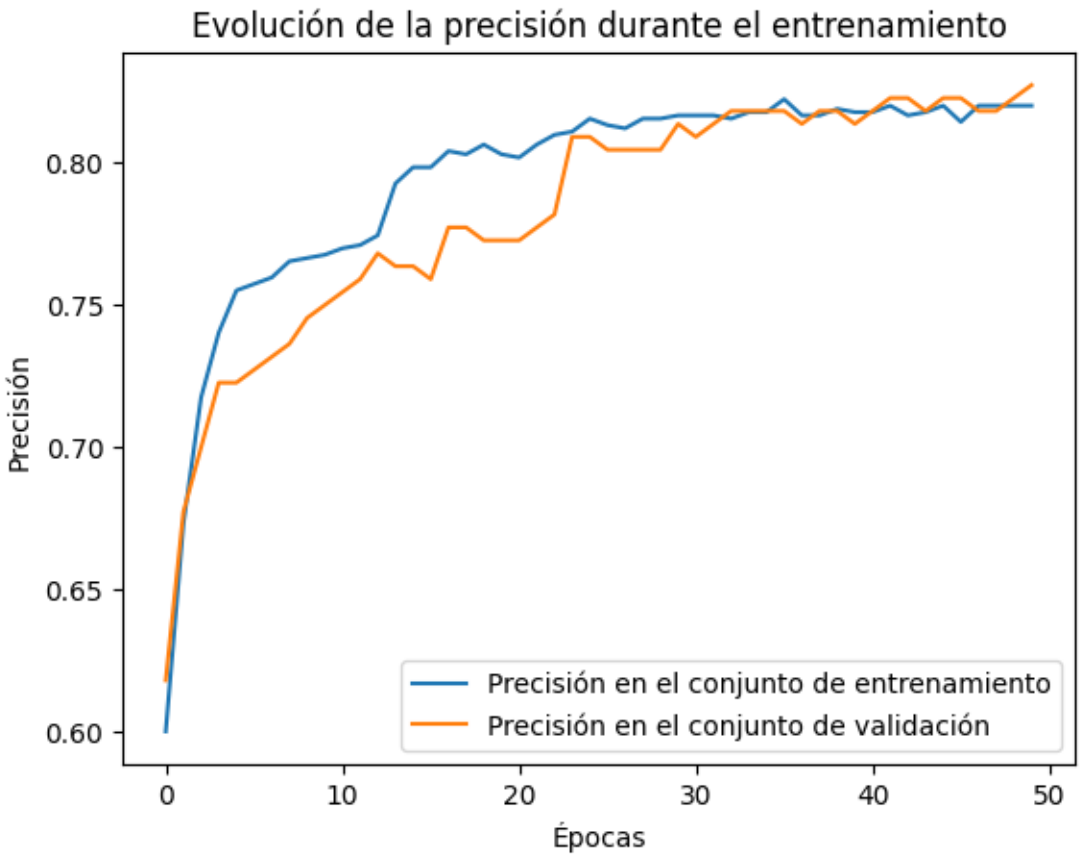
4. Línea Naranja (Pérdida en el Conjunto de Validación):

- Muestra la evolución de la pérdida en el conjunto de validación.
- La pérdida en el conjunto de validación también disminuye, aunque no tan rápido como en el conjunto de entrenamiento.

*Observaciones:*

- Ambas líneas muestran una tendencia a la baja, lo que indica que el modelo está aprendiendo a minimizar el error tanto en el conjunto de entrenamiento como en el de validación.
- Al final del entrenamiento, la pérdida se estabiliza, lo que sugiere que el modelo ha alcanzado un punto donde no hay mejoras significativas adicionales.
- La diferencia entre la pérdida de entrenamiento y validación no es muy grande, lo que sugiere que el modelo no está sobreajustando significativamente.

[ ]:
```
# Graficar la precisión durante el entrenamiento
plt.plot(history.history['accuracy'], label='Precisión en el conjunto de␣
 ↪entrenamiento')
plt.plot(history.history['val_accuracy'], label='Precisión en el conjunto de␣
 ↪validación')
plt.xlabel('Épocas')
plt.ylabel('Precisión')
plt.legend()
plt.title('Evolución de la precisión durante el entrenamiento')
plt.show()
```




1. Eje X (Épocas):

- Representa el número de épocas (iteraciones) durante el entrenamiento del modelo.

2. Eje Y (Precisión):

- Representa el valor de la precisión calculada en cada época.

3. Línea Azul (Precisión en el Conjunto de Entrenamiento):

- Muestra la evolución de la precisión en el conjunto de entrenamiento.
- La precisión aumenta rápidamente al inicio, lo que indica que el modelo está aprendiendo a clasificar correctamente los datos de entrenamiento.

4. Línea Naranja (Precisión en el Conjunto de Validación):

- Muestra la evolución de la precisión en el conjunto de validación.
- La precisión en el conjunto de validación también aumenta y se mantiene cercana a la precisión del conjunto de entrenamiento.

*Observaciones:*

- Ambas líneas muestran una tendencia al alza, lo que indica que el modelo está mejorando su capacidad para hacer predicciones correctas.

- La precisión en el conjunto de validación es ligeramente menor que en el conjunto de entrenamiento, lo cual es esperado, pero ambas precisiones se estabilizan en valores altos (>0.80).
- La falta de divergencia significativa entre las líneas azul y naranja sugiere que el modelo generaliza bien y no está sobreajustando.

**Hacemos predicciones y evaluamos con métricas de clasificación**

[ ]:
```
from sklearn.metrics import confusion_matrix, classification_report
```

[ ]:
```
# Hacer predicciones en el conjunto de prueba
y_pred = (model.predict(X_test) > 0.5).astype("int32")
```

```
7/7 [==============================] - 0s 3ms/step
```

[ ]:
```
# Matriz de confusión
conf_matrix = confusion_matrix(y_test, y_pred)
print('Matriz de Confusión:')
print(conf_matrix)
```

```
Matriz de Confusión:
[[95 19]
 [19 87]]
```

[ ]:
```
# Reporte de clasificación
class_report = classification_report(y_test, y_pred)
print('Reporte de Clasificación:')
print(class_report)
```

```
Reporte de Clasificación:
              precision    recall  f1-score   support

           0       0.83      0.83      0.83       114
           1       0.82      0.82      0.82       106

    accuracy                           0.83       220
   macro avg       0.83      0.83      0.83       220
weighted avg       0.83      0.83      0.83       220
```

**Interpretación del Reporte de Clasificación**

El reporte de clasificación nos ofrece una evaluación detallada del rendimiento del modelo en la tarea de predicción de la supervivencia de los pasajeros del Titanic. Las métricas claves que se muestran son precisión (precision), recall (recuperación) y F1-score para cada clase, así como la precisión general (accuracy).

Métricas por Clase

1. Clase 0 (No sobrevivió):

- Precisión (Precision): 0.83

- La precisión indica el porcentaje de predicciones correctas entre todas las predicciones hechas para la clase 0 (no sobrevivió). En este caso, el 83% de las predicciones de que un pasajero no sobrevivió fueron correctas.
- Recall (Recuperación): 0.83
- El recall mide el porcentaje de verdaderos positivos que fueron correctamente identificados. Aquí, el 83% de los pasajeros que realmente no sobrevivieron fueron correctamente identificados por el modelo.
- F1-score: 0.83
- El F1-score es la media armónica entre la precisión y el recall, proporcionando una medida balanceada del rendimiento del modelo para esta clase. Un F1-score de 0.83 indica un buen equilibrio entre precisión y recall.

2. Clase 1 (Sobrevivió):

- Precisión (Precision): 0.82
- La precisión para la clase 1 indica que el 82% de las predicciones de que un pasajero sobrevivió fueron correctas.
- Recall (Recuperación): 0.82
- El recall para la clase 1 indica que el 82% de los pasajeros que realmente sobrevivieron fueron correctamente identificados por el modelo.
- F1-score: 0.82
- Un F1-score de 0.82 para la clase 1 indica que el modelo mantiene un buen equilibrio entre precisión y recall para los pasajeros que sobrevivieron.

*Métricas Globales*

- Precisión General (Accuracy): 0.83
- La precisión general del modelo es del 83%, lo que significa que el modelo clasifica correctamente el 83% de las muestras en el conjunto de prueba.
- Macro Promedio (Macro Avg):
- Precisión: 0.83
- Recall: 0.83
- F1-score: 0.83
- El macro promedio es el promedio no ponderado de las métricas para todas las clases. Indica el rendimiento medio del modelo en cada clase sin considerar el desequilibrio de clases.
- Promedio Ponderado (Weighted Avg):
- Precisión: 0.83
- Recall: 0.83
- F1-score: 0.83
- El promedio ponderado considera el número de muestras en cada clase. En este caso, las métricas ponderadas coinciden con el macro promedio porque las clases están razonablemente balanceadas.

**Conclusión**

En resumen, el modelo de clasificación muestra un rendimiento sólido con una precisión, recall y F1-score equilibrados en ambas clases. La precisión general del 83% indica que el modelo es efectivo para predecir la supervivencia de los pasajeros del Titanic. Las métricas equilibradas sugieren que el modelo maneja bien tanto las predicciones de supervivencia como las de no supervivencia sin un sesgo significativo hacia una clase u otra.