Notebook creado por **Guillermo Grande Santi**

Imports

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import logging
import math
from collections import Counter
from scipy.stats import norm

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, roc_auc_score
from sklearn.model_selection import GridSearchCV, cross_val_score
import pickle

import tensorflow as tf
from tensorflow.keras.models import Sequential # type: ignore
from tensorflow.keras.layers import LSTM, Dense # type: ignore
from tensorflow.keras.preprocessing.sequence import pad_sequences #
type: ignore
from tensorflow.keras.preprocessing.text import Tokenizer # type:
ignore

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import torch.backends.cudnn as cudnn

from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModel
from gensim.models import Word2Vec
from gensim.utils import simple_preprocess
import nltk
import re
import string
import spacy
import contractions

import shap
```

```
c:\Users\guigr\anaconda3\envs\tfm\Lib\site-packages\tqdm\auto.py:21:
TqdmWarning: IProgress not found. Please update jupyter and
```

```
ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm

WARNING:tensorflow:From C:\Users\guigr\AppData\Roaming\Python\
Python311\site-packages\tf_keras\src\losses.py:2976: The name
tf.losses.sparse_softmax_cross_entropy is deprecated. Please use
tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

Vectorización TF-IDF + Redes Neuronales MLPs

Aunque el Random Forest ha logrado una precisión del 98.4% en la detección de noticias falsas, es valioso explorar el desempeño de **redes neuronales**, ya que pueden capturar patrones complejos en los datos.

En primer lugar, utilizaremos la misma vectorización TF-IDF, evaluaremos si modelos neuronales ofrecen ventajas adicionales en términos de generalización y robustez.

Más tarde, probaremos con otras **formas de vectorización**.

Para terminar, exploraremos si **redes neuronales recurrentes** (RNNs) ofrecen un rendimiento superior al capturar dependencias secuenciales y contextuales en los textos.

División en Train, Validation & Test para Redes Neuronales

```
# Cargar el DataFrame limpio
df = pd.read_csv("../Datasets/Cleaned-FR-News_V2.csv")

# Dividimos los datos en entrenamiento y prueba
# Por ahora usaremos únicamente el texto de la noticia (omitimos el
título)
X = df["clean_text"]
y = df["label"]
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Se usará para redes neuronales
# Usaremos un 20% del conjunto de datos para validación (16% del
total)
X_train, X_valid, y_train, y_valid = train_test_split(X_train,
y_train, test_size=0.2, random_state=42)

print("Shape of X_train:", X_train.shape)
print("Shape of X_valid:", X_valid.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_valid:", y_valid.shape)
print("Shape of y_test:", y_test.shape)
```

```
Shape of X_train: (28283,)
Shape of X_valid: (7071,)
Shape of X_test: (8839,)
Shape of y_train: (28283,)
Shape of y_valid: (7071,)
Shape of y_test: (8839,)
```

Vectorización mediante TF-IDF + Red Neuronal MLP

Tokenización y Tensorización

```
# Definir y ajustar el vectorizador TF-IDF en el conjunto de entrenamiento
vectorizer = TfidfVectorizer(max_features=5000)
X_train = vectorizer.fit_transform(X_train)

# Transformar los conjuntos de validación y prueba utilizando el vectorizador ajustado
X_valid = vectorizer.transform(X_valid)
X_test = vectorizer.transform(X_test)

# Verificar las dimensiones de los conjuntos transformados
print("Shape of X_train_tfidf:", X_train.shape)
print("Shape of X_valid_tfidf:", X_valid.shape)
print("Shape of X_test_tfidf:", X_test.shape)
```

```
Shape of X_train_tfidf: (28283, 5000)
Shape of X_valid_tfidf: (7071, 5000)
Shape of X_test_tfidf: (8839, 5000)
```

Los **tensores** son estructuras de datos multidimensionales similares a matrices que pueden representar datos en múltiples dimensiones.

En **PyTorch**, los tensores son el núcleo de su funcionalidad, ya que permiten realizar operaciones matemáticas eficientes y aprovechar la aceleración por hardware, como GPUs.

Es necesario **convertir los datos a tensores** en PyTorch porque este formato es compatible con las operaciones optimizadas de la biblioteca, como el cálculo de gradientes automáticos y el entrenamiento de modelos.

```
# Convertir datos a tensores de PyTorch
def tensorize(data, labels, device):
    data_tensor = torch.tensor(data.toarray(), dtype=torch.float32).to(device)
    labels_tensor = torch.tensor(labels.values, dtype=torch.float32).unsqueeze(1).to(device)
    return data_tensor, labels_tensor

# Verificar si la GPU está disponible
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
print("Using device:", device)

# Tensorizar los conjuntos de datos
X_train_tensor, y_train_tensor = tensorize(X_train, y_train, device)
X_valid_tensor, y_valid_tensor = tensorize(X_valid, y_valid, device)
X_test_tensor, y_test_tensor = tensorize(X_test, y_test, device)

print("Shape of X_train_tensor:", X_train_tensor.shape)
print("Shape of y_train_tensor:", y_train_tensor.shape)
print("Shape of X_valid_tensor:", X_valid_tensor.shape)
print("Shape of y_valid_tensor:", y_valid_tensor.shape)
```

```
Using device: cuda
Shape of X_train_tensor: torch.Size([28283, 5000])
Shape of y_train_tensor: torch.Size([28283, 1])
Shape of X_valid_tensor: torch.Size([7071, 5000])
Shape of y_valid_tensor: torch.Size([7071, 1])
```

```
# Create DataLoaders para entrenamiento y validación
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
valid_dataset = TensorDataset(X_valid_tensor, y_valid_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# Variables de configuración
batch_size = 128

# Crear DataLoaders para entrenamiento y validación
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size,
shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)
```

Definición del modelo

```
# Definimos modelo MLP
mlp_model = nn.Sequential(
    nn.Linear(5000, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(64, 1),
    nn.Sigmoid()
)
```

Funciones para entrenamiento y validación

```
def train_model(model, train_loader, valid_loader, criterion,
optimizer, num_epochs, device, best_model_path):
```

```
    """
    Entrena el modelo MLP y guarda el mejor modelo basado en la
pérdida de validación.

    Args:
        model: El modelo MLP a entrenar.
        train_loader: DataLoader para el conjunto de entrenamiento.
        valid_loader: DataLoader para el conjunto de validación.
        criterion: Función de pérdida.
        optimizer: Optimizador.
        num_epochs: Número de épocas para entrenar.
        device: Dispositivo (CPU o GPU).
        best_model_path: Ruta para guardar el mejor modelo.
    """
    train_losses, valid_losses = [], []
    train_accuracies, valid_accuracies = [], []
    best_val_loss = float("inf")

    # Entrenamiento
    for epoch in range(num_epochs):
        model.train()
        train_loss, correct_train, total_train = 0, 0, 0

        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)

            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            predicted = (outputs >= 0.5).float()
            correct_train += (predicted == y_batch).sum().item()
            total_train += y_batch.size(0)

        train_losses.append(train_loss / len(train_loader))
        train_accuracies.append(correct_train / total_train)

        # Validation
        model.eval()
        valid_loss, correct_valid, total_valid = 0, 0, 0

        with torch.no_grad():
            for X_batch, y_batch in valid_loader:
                X_batch, y_batch = X_batch.to(device),
y_batch.to(device)
                outputs = model(X_batch)
                loss = criterion(outputs, y_batch)
```

```
                valid_loss += loss.item()
                predicted = (outputs >= 0.5).float()
                correct_valid += (predicted == y_batch).sum().item()
                total_valid += y_batch.size(0)

        valid_losses.append(valid_loss / len(valid_loader))
        valid_accuracies.append(correct_valid / total_valid)

        # Guardar el mejor modelo
        if valid_losses[-1] < best_val_loss:
            best_val_loss = valid_losses[-1]
            torch.save(model.state_dict(), best_model_path)
            print(f"□ Saved best model at {best_model_path} (Val Loss:
{best_val_loss:.4f})")

        print(f"Epoch [{epoch+1}/{num_epochs}] - Loss: {train_losses[-
1]:.4f} - Acc: {train_accuracies[-1]:.4f} - Val Loss: {valid_losses[-
1]:.4f} - Val Acc: {valid_accuracies[-1]:.4f}")

    return train_losses, valid_losses, train_accuracies,
valid_accuracies

def plot_loss_and_accuracy(train_losses, valid_losses,
train_accuracies, valid_accuracies):
    """
    Función para graficar la pérdida y precisión de entrenamiento y
validación a lo largo de las épocas.

    Args:
    - train_losses: Lista de pérdidas de entrenamiento para cada
época.
    - valid_losses: Lista de pérdidas de validación para cada época.
    - train_accuracies: Lista de precisiones de entrenamiento para
cada época.
    - valid_accuracies: Lista de precisiones de validación para cada
época.
    """
    plt.figure(figsize=(12, 5))

    # Plot Loss
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label="Train Loss")
    plt.plot(valid_losses, label="Val Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Loss over epochs")
    plt.legend()

    # Plot Accuracy
    plt.subplot(1, 2, 2)
```

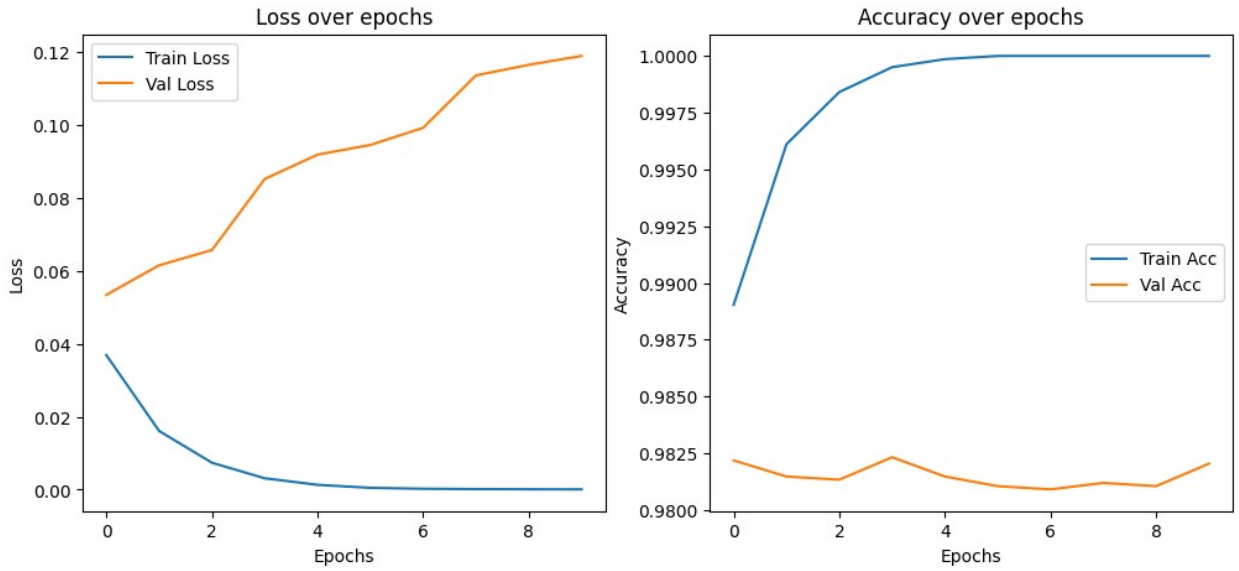
```
    plt.plot(train_accuracies, label="Train Acc")
    plt.plot(valid_accuracies, label="Val Acc")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.title("Accuracy over epochs")
    plt.legend()

    plt.show()

def evaluate_model(model, test_loader, device):
    """
    Evalúa el modelo en el conjunto de prueba utilizando un DataLoader
para evitar problemas de memoria.

    Parámetros:
    - model: El modelo PyTorch entrenado.
    - test_loader: DataLoader para el conjunto de prueba.
    - device: El dispositivo (CPU/GPU) a usar para el cálculo.
    """
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            X_batch = X_batch.to(device)
            outputs = model(X_batch)
            preds = (outputs >= 0.5).float().cpu().numpy()
            all_preds.extend(preds)
            all_labels.extend(y_batch.cpu().numpy())

    # Calculate accuracy
    test_accuracy = accuracy_score(all_labels, all_preds)
    print(f"Precisión en el conjunto de prueba: {test_accuracy:.4f}")
    auc_roc = roc_auc_score(all_labels, all_preds)
    print(f"AUC-ROC en el conjunto de prueba: {auc_roc:.4f}")
```

Entrenamiento y validación

```
# Instanciar modelo y moverlo a GPU si está disponible
model = mlp_model.to(device)

# Función de pérdida y optimizador
criterion = nn.BCELoss()  # Pérdida para clasificación binaria
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Variables de ntrenamiento
num_epochs = 10
train_losses, valid_losses = [], []
train_accuracies, valid_accuracies = [], []
```

```python
# Ruta donde se guardará el mejor modelo
best_model_path = "../models/best_mlp.pth"

# Entrenar el modelo
train_losses, valid_losses, train_accuracies, valid_accuracies =
train_model(
    model, train_loader, valid_loader, criterion, optimizer,
num_epochs, device, best_model_path
)

# Graficar pérdidas y precisiones
plot_loss_and_accuracy(train_losses, valid_losses, train_accuracies,
valid_accuracies)

# Evaluar el modelo en el conjunto de prueba
evaluate_model(model, test_loader, device)
```

```
 Saved best model at ../models/Test.pth (Val Loss: 0.0534)
Epoch [1/10] - Loss: 0.0369 - Acc: 0.9890 - Val Loss: 0.0534 - Val
Acc: 0.9822
Epoch [2/10] - Loss: 0.0161 - Acc: 0.9961 - Val Loss: 0.0616 - Val
Acc: 0.9815
Epoch [3/10] - Loss: 0.0074 - Acc: 0.9984 - Val Loss: 0.0657 - Val
Acc: 0.9813
Epoch [4/10] - Loss: 0.0031 - Acc: 0.9995 - Val Loss: 0.0852 - Val
Acc: 0.9823
Epoch [5/10] - Loss: 0.0013 - Acc: 0.9999 - Val Loss: 0.0919 - Val
Acc: 0.9815
Epoch [6/10] - Loss: 0.0005 - Acc: 1.0000 - Val Loss: 0.0946 - Val
Acc: 0.9810
Epoch [7/10] - Loss: 0.0003 - Acc: 1.0000 - Val Loss: 0.0993 - Val
Acc: 0.9809
Epoch [8/10] - Loss: 0.0002 - Acc: 1.0000 - Val Loss: 0.1136 - Val
Acc: 0.9812
Epoch [9/10] - Loss: 0.0001 - Acc: 1.0000 - Val Loss: 0.1165 - Val
Acc: 0.9810
Epoch [10/10] - Loss: 0.0001 - Acc: 1.0000 - Val Loss: 0.1190 - Val
Acc: 0.9820
```




```
Precisión en el conjunto de prueba: 0.9850
AUC-ROC en el conjunto de prueba: 0.9850
```

El modelo ha mostrado un rendimiento sobresaliente, alcanzando una precisión en validación cercana al 98.2% y una pérdida de 0.05 tras solo **tres épocas**.

En el conjunto de prueba, se obtuvo el mejor desempeño con una precisión del **98.5%**.

Vectorización mediante Embeddings (*BERT*) + Red Neuronal MLP

En lugar de utilizar **TF-IDF** como vectorización inicial, vamos a probar con embeddings como **BERT**, pues podría ser una excelente idea al capturar mejor el significado semántico del texto completo, a diferencia de TF-IDF, que se basa en la frecuencia de palabras y carece de contexto.

Además, BERT reduce significativamente la **dimensionalidad** (768 vs 5000), al mismo tiempo que captura relaciones más profundas en el texto, lo que puede optimizar la eficiencia computacional y reducir el riesgo de sobreajuste.

No obstante, hay que destacar que al usar **sentence-transformers** se obtiene una **única representación fija para cada noticia**, lo que sigue haciendo incompatible el uso de LSTMs, ya que no se trabaja con secuencias de vectores por palabra, sino con un solo embedding por entrada.

Vectorización mediante *Sentence-transformers*

El model de embeddings que se va a utilizar, **bert-base-nli-mean-tokens**, presenta las siguientes características:

- **`bert-base`** → Está basado en el modelo BERT con 12 capas, 768 dimensiones ocultas y 12 cabezas de atención.

- **`nli` (Inferencia de Lenguaje Natural)** → El modelo fue ajustado con datasets de NLI (SNLI + MNLI) para entender relaciones semánticas entre oraciones.
- **`mean-tokens`** → Utiliza un promedio de los embeddings de todos los tokens para crear embeddings de oraciones, en lugar de usar solo el token `[CLS]`.

```
# Cargar el DataFrame limpio
df = pd.read_csv("../Datasets/Cleaned-FR-News_V2.csv")

# Dividimos los datos en entrenamiento y prueba
# Por ahora usaremos únicamente el texto de la noticia (omitimos el título)
X = df["clean_text"]
y = df["label"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Se usará para redes neuronales
# Usaremos un 20% del conjunto de datos para validación (16% del total)
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

model = SentenceTransformer("bert-base-nli-mean-tokens") # Modelo rápido

# Convertir a embeddings
X_train_embedding = model.encode(X_train.tolist())
X_valid_embedding = model.encode(X_valid.tolist())
X_test_embedding = model.encode(X_test.tolist())

# Verificar las dimensiones de los conjuntos transformados
print("Shape of X_train_tfidf:", X_train.shape)
print("Shape of X_valid_tfidf:", X_valid.shape)
print("Shape of X_test_tfidf:", X_test.shape)
```

```
(28283, 768)
```

Entrenamiento y evaluación

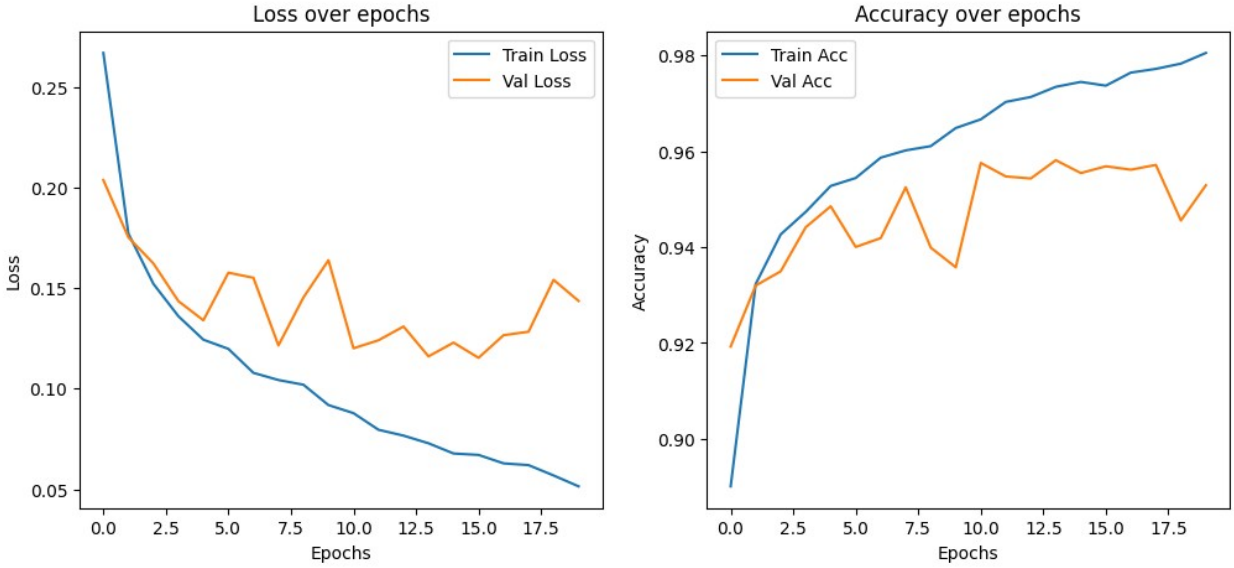
```
# GPU disponible
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Convertir datos a tensores de PyTorch cuando ya son arrays
def tensorize_array(data, labels, device):
    data_tensor = torch.tensor(data, dtype=torch.float32).to(device)
    labels_tensor = torch.tensor(labels.values, dtype=torch.float32).unsqueeze(1).to(device)
    return data_tensor, labels_tensor

X_train_tensor, y_train_tensor = tensorize_array(X_train_embedding, y_train, device)
```

```
X_valid_tensor, y_valid_tensor = tensorize_array(X_valid_embedding,
y_valid, device)
X_test_tensor, y_test_tensor = tensorize_array(X_test_embedding,
y_test, device)

# Create DataLoaders para entrenamiento y validación
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
valid_dataset = TensorDataset(X_valid_tensor, y_valid_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# Variables de configuración
batch_size = 128

train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size,
shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

# Definimos modelo MLP
mlp_model = nn.Sequential(
    nn.Linear(768, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(64, 1),
    nn.Sigmoid()
)

# Instanciar modelo y moverlo a GPU si está disponible
model = mlp_model.to(device)

# Función de pérdida y optimizador
criterion = nn.BCELoss()  # Pérdida para clasificación binaria
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Variables de ntrenamiento
num_epochs = 20
train_losses, valid_losses = [], []
train_accuracies, valid_accuracies = [], []

# Ruta donde se guardará el mejor modelo
best_model_path = "../models/best_mlp_embeddings.pth"

# Entrenar el modelo
train_losses, valid_losses, train_accuracies, valid_accuracies =
train_model(
    model, train_loader, valid_loader, criterion, optimizer,
```

```
num_epochs, device, best_model_path
)

# Graficar pérdidas y precisiones
plot_loss_and_accuracy(train_losses, valid_losses, train_accuracies,
valid_accuracies)

# Evaluar el modelo en el conjunto de prueba
evaluate_model(model, test_loader, device)
```

```
 Saved best model at models/best_mlp_embeddings.pth (Val Loss:
0.2037)
Epoch [1/20] - Loss: 0.2669 - Acc: 0.8901 - Val Loss: 0.2037 - Val
Acc: 0.9192
 Saved best model at models/best_mlp_embeddings.pth (Val Loss:
0.1752)
Epoch [2/20] - Loss: 0.1771 - Acc: 0.9323 - Val Loss: 0.1752 - Val
Acc: 0.9320
 Saved best model at models/best_mlp_embeddings.pth (Val Loss:
0.1622)
Epoch [3/20] - Loss: 0.1521 - Acc: 0.9427 - Val Loss: 0.1622 - Val
Acc: 0.9349
 Saved best model at models/best_mlp_embeddings.pth (Val Loss:
0.1435)
Epoch [4/20] - Loss: 0.1360 - Acc: 0.9474 - Val Loss: 0.1435 - Val
Acc: 0.9441
 Saved best model at models/best_mlp_embeddings.pth (Val Loss:
0.1340)
Epoch [5/20] - Loss: 0.1243 - Acc: 0.9528 - Val Loss: 0.1340 - Val
Acc: 0.9485
Epoch [6/20] - Loss: 0.1198 - Acc: 0.9544 - Val Loss: 0.1576 - Val
Acc: 0.9400
Epoch [7/20] - Loss: 0.1078 - Acc: 0.9587 - Val Loss: 0.1551 - Val
Acc: 0.9419
 Saved best model at models/best_mlp_embeddings.pth (Val Loss:
0.1215)
Epoch [8/20] - Loss: 0.1043 - Acc: 0.9602 - Val Loss: 0.1215 - Val
Acc: 0.9525
Epoch [9/20] - Loss: 0.1019 - Acc: 0.9611 - Val Loss: 0.1453 - Val
Acc: 0.9399
Epoch [10/20] - Loss: 0.0918 - Acc: 0.9649 - Val Loss: 0.1638 - Val
Acc: 0.9358
 Saved best model at models/best_mlp_embeddings.pth (Val Loss:
0.1201)
Epoch [11/20] - Loss: 0.0878 - Acc: 0.9667 - Val Loss: 0.1201 - Val
Acc: 0.9576
Epoch [12/20] - Loss: 0.0796 - Acc: 0.9703 - Val Loss: 0.1240 - Val
Acc: 0.9547
Epoch [13/20] - Loss: 0.0767 - Acc: 0.9713 - Val Loss: 0.1309 - Val
Acc: 0.9543
```

```
 Saved best model at models/best_mlp_embeddings.pth (Val Loss:
0.1160)
Epoch [14/20] - Loss: 0.0728 - Acc: 0.9734 - Val Loss: 0.1160 - Val
Acc: 0.9581
Epoch [15/20] - Loss: 0.0678 - Acc: 0.9745 - Val Loss: 0.1229 - Val
Acc: 0.9555
 Saved best model at models/best_mlp_embeddings.pth (Val Loss:
0.1153)
Epoch [16/20] - Loss: 0.0671 - Acc: 0.9737 - Val Loss: 0.1153 - Val
Acc: 0.9569
Epoch [17/20] - Loss: 0.0629 - Acc: 0.9764 - Val Loss: 0.1266 - Val
Acc: 0.9562
Epoch [18/20] - Loss: 0.0620 - Acc: 0.9772 - Val Loss: 0.1283 - Val
Acc: 0.9571
Epoch [19/20] - Loss: 0.0569 - Acc: 0.9783 - Val Loss: 0.1541 - Val
Acc: 0.9456
Epoch [20/20] - Loss: 0.0514 - Acc: 0.9805 - Val Loss: 0.1436 - Val
Acc: 0.9529
```




```
Precisión en el conjunto de prueba: 0.9628
```

No se observa un rendimiento superior comparando con el modelo que utilizaba vectorización TF-IDF como entrada de la MLP, obteniendo un **96.3%** de precisión en el conjunto de prueba.

La **MLP** con **TF-IDF** podría funcionar mejor en la clasificación de noticias porque TF-IDF enfatiza **términos clave** específicos de cada noticia, lo que ayuda a **diferenciar mejor** las categorías. En cambio, **BERT**, al generar embeddings densos y compactos de 768 dimensiones, puede perder detalles relevantes al **comprimir la información** en un espacio más reducido.

Vectorización mediante Embeddings + LSTM

La clasificación de fake news utilizando una **RNN (Recurrent Neural Network)** es una excelente idea debido a su capacidad para manejar **secuencias de texto** largas y complejas, lo cual es fundamental en el análisis de noticias.

Las LSTM son especialmente efectivas para **capturar dependencias a largo plazo** en los datos, lo que permite identificar patrones y contextos importantes en las noticias, incluso cuando la información relevante está dispersa a lo largo del texto.

Además, se escoge una red **LSTM (Long Short-Term Memory)** por su capacidad para mitigar el problema del desvanecimiento del gradiente, que afecta a otras redes neuronales recurrentes, lo que las hace muy adecuadas para procesar y clasificar textos de forma eficiente y precisa, mejorando la detección de desinformación en el proceso.

División en Train, Validation & Test para Redes Neuronales

```
# Cargar el DataFrame limpio
df = pd.read_csv("../Datasets/Cleaned-FR-News_V2.csv")

# Dividimos los datos en entrenamiento y prueba
# Por ahora usaremos únicamente el texto de la noticia (omitimos el título)
X = df["clean_text"]
y = df["label"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Se usará para redes neuronales
# Usaremos un 20% del conjunto de datos para validación (16% del total)
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

print("Shape of X_train:", X_train.shape)
print("Shape of X_valid:", X_valid.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_valid:", y_valid.shape)
print("Shape of y_test:", y_test.shape)
```

```
Shape of X_train: (28283,)
Shape of X_valid: (7071,)
Shape of X_test: (8839,)
Shape of y_train: (28283,)
Shape of y_valid: (7071,)
Shape of y_test: (8839,)
```

Vectorización con Tokenizer + LSTM Básica

Tokenización y Tensorización

En cuanto a la vectorización de nuestras noticias, es importante destacar que **TF-IDF** se enfoca en ponderar la importancia de las palabras en los documentos, pero **no captura la secuencia temporal** ni el contexto de las palabras en el texto, lo que puede limitar el rendimiento en modelos que dependen de la información secuencial, como las LSTM.

Por esta razón, utilizaremos un tokenizador que **preserve el orden temporal**. En este caso, emplearemos el **Tokenizer** de *TensorFlow*, que convierte el texto en secuencias de índices enteros que representan palabras, manteniendo tanto el orden como la estructura temporal del texto.

El proceso que realiza **Tokenizer** es el siguiente:

- **Tokenización**: Primero, las noticias se dividen en palabras o subpalabras (*tokens*), y cada palabra se asigna a un número único (*índice*) dentro del vocabulario.
- **Vectorización**: El *Tokenizer* convierte cada noticia en una secuencia de índices que corresponden a las palabras en el vocabulario. Por ejemplo, como nuestro vocabulario tiene un tamaño de 5000 palabras, cada palabra en el texto se mapea a un número entre 0 y 4999.

Además, cada noticia, una vez convertida en una secuencia de índices, será transformada en una secuencia de exactamente **1256 elementos**. Esto se hace porque, de media, las noticias recogidas tenían una longitud media de 1656 palabras tras el preprocesado. Sin embargo, un número de elementos mayor podría empeorar el rendimiento del entrenamiento.

Todas las secuencias tendrán la misma longitud gracias a la función **pad_sequences** de *Tensorflow*, pues realiza dos acciones dependiendo de la longitud de la secuencia:

- Si la noticia tiene menos de 1256 palabras, se agrega padding (relleno) con ceros.
- Si la noticia tiene más de 1256 palabras, se trunca la secuenia para que tenga solo 1256 índices.

Por defecto, el padding y el truncado se realizan al inicio.

```
# --------------------
# Tokenización y secuencias
# --------------------
# Disponibilidad de GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando device: {device}")

# Número máximo de palabras a considerar en el vocabulario
max_words = 5000
# Longitud máxima de las secuencias - La media de longitud de los
textos es 1656 aproximadamente
max_len = 1256
```

```
# Inicializar el tokenizador de TensorFlow con un vocabulario limitado
y un token para palabras 'Out-Of-Vocabulary'
tokenizer = Tokenizer(num_words=max_words, oov_token='<OOV>')
# Ajustar el tokenizador al texto de entrenamiento
tokenizer.fit_on_texts(X_train)

# Convertir textos a secuencias de índices y aplicar padding
X_train_embedding = tokenizer.texts_to_sequences(X_train)
X_train_embedding = pad_sequences(X_train_embedding, maxlen=max_len)
X_valid_embedding = tokenizer.texts_to_sequences(X_valid)
X_valid_embedding = pad_sequences(X_valid_embedding, maxlen=max_len)
X_test_embedding = tokenizer.texts_to_sequences(X_test)
X_test_embedding = pad_sequences(X_test_embedding, maxlen=max_len)

# Convertir a tensores de PyTorch --- Pasamos a device en el bucle de
entrenamiento
X_train_tensor = torch.tensor(X_train_embedding, dtype=torch.long)
y_train_tensor = torch.tensor(y_train.values,
dtype=torch.float32).unsqueeze(1)
X_valid_tensor = torch.tensor(X_valid_embedding, dtype=torch.long)
y_valid_tensor = torch.tensor(y_valid.values,
dtype=torch.float32).unsqueeze(1)
X_test_tensor = torch.tensor(X_test_embedding, dtype=torch.long)
y_test_tensor = torch.tensor(y_test.values,
dtype=torch.float32).unsqueeze(1)

print("Shape of X_train_tensor:", X_train_tensor.shape)
print("Shape of y_train_tensor:", y_train_tensor.shape)
print("Shape of X_valid_tensor:", X_valid_tensor.shape)
print("Shape of y_valid_tensor:", y_valid_tensor.shape)
print("Shape of X_test_tensor:", X_test_tensor.shape)
print("Shape of y_test_tensor:", y_test_tensor.shape)

# Batch Size
batch_size = 128

# Crear Datasets & DataLoaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
valid_dataset = TensorDataset(X_valid_tensor, y_valid_tensor)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size,
shuffle=False)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

Usando device: cuda
Shape of X_train_tensor: torch.Size([28283, 1256])
```

```
Shape of y_train_tensor: torch.Size([28283, 1])
Shape of X_valid_tensor: torch.Size([7071, 1256])
Shape of y_valid_tensor: torch.Size([7071, 1])
Shape of X_test_tensor: torch.Size([8839, 1256])
Shape of y_test_tensor: torch.Size([8839, 1])
```

Definición del modelo

Para que el modelo **LSTM** funcione correctamente, es necesario que los datos de entrada tengan **tres dimensiones**. Actualmente, el tensor de entrenamiento tiene una forma de *torch.Size([28283, 1256])*, es decir, contiene secuencias de longitud 1256 para 28,283 muestras, pero aún le falta la dimensión correspondiente a la representación vectorial de cada palabra.

Esta tercera dimensión se añade mediante la capa de **Embedding**, la cual transforma cada índice entero (asociado a una palabra) en un **vector denso de características** continuas. Por ejemplo, si el índice 23 representa la palabra "economía", el Embedding devolverá un vector de dimensión 48 que captura sus propiedades semánticas. Así, el modelo puede aprender relaciones y patrones complejos entre palabras en función de su contexto, en lugar de tratarlas simplemente como identificadores numéricos.

Una vez procesadas las secuencias por la LSTM, se toma el **último estado oculto de la salida** (lstm_out[:, -1, :]), el cual resume la información contextual de toda la secuencia. Este vector condensado se pasa luego por una capa lineal para realizar la **clasificación binaria** entre noticias reales y falsas.

```
# --------------------
# Modelo LSTM
# --------------------
class LSTMModel(nn.Module):
    def __init__(self, input_dim=max_words, embedding_dim=48,
hidden_dim=64):
        super(LSTMModel, self).__init__()
        # Capa de Embeddings
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        # Capa LSTM
        self.lstm = nn.LSTM(embedding_dim, hidden_dim,
batch_first=True)
        # Dropout para regularización
        self.dropout = nn.Dropout(p=0.3)
        # Capa densa final con activación sigmoide
        self.fc = nn.Linear(hidden_dim, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)  # Embeddings -> (batch_size, max_len,
embedding_dim) = (128, 1256, 48)
        lstm_out, _ = self.lstm(x)  # Pasamos por la LSTM
        lstm_out = self.dropout(lstm_out[:, -1, :])  # Tomamos solo el
último estado, capturando el contexto completo de la secuencia de
```

```
entrada para hacer la clasificación
        out = self.fc(lstm_out)  # Pasamos por la capa densa (64 → 1)
        return self.sigmoid(out)  # Activación sigmoide
```

Funciones para entrenamiento y evaluación

```
def train_model(model, train_loader, valid_loader, criterion,
optimizer, num_epochs, device, best_model_path):
    """
    Entrena el modelo LSTM y guarda el mejor modelo basado en la
pérdida de validación.

    Args:
        model: El modelo MLP a entrenar.
        train_loader: DataLoader para el conjunto de entrenamiento.
        valid_loader: DataLoader para el conjunto de validación.
        criterion: Función de pérdida.
        optimizer: Optimizador.
        num_epochs: Número de épocas para entrenar.
        device: Dispositivo (CPU o GPU).
        best_model_path: Ruta para guardar el mejor modelo.
    """
    train_losses, valid_losses = [], []
    train_accuracies, valid_accuracies = [], []
    best_val_loss = float("inf")

    # Entrenamiento
    for epoch in range(num_epochs):
        model.train()
        train_loss, correct_train, total_train = 0, 0, 0

        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)

            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            predicted = (outputs >= 0.5).float()
            correct_train += (predicted == y_batch).sum().item()
            total_train += y_batch.size(0)

        train_losses.append(train_loss / len(train_loader))
        train_accuracies.append(correct_train / total_train)

        # Validation
        model.eval()
```

```
        valid_loss, correct_valid, total_valid = 0, 0, 0

        with torch.no_grad():
            for X_batch, y_batch in valid_loader:
                X_batch, y_batch = X_batch.to(device),
y_batch.to(device)
                outputs = model(X_batch)
                loss = criterion(outputs, y_batch)
                valid_loss += loss.item()
                predicted = (outputs >= 0.5).float()
                correct_valid += (predicted == y_batch).sum().item()
                total_valid += y_batch.size(0)

        valid_losses.append(valid_loss / len(valid_loader))
        valid_accuracies.append(correct_valid / total_valid)

        # Guardar el mejor modelo
        if valid_losses[-1] < best_val_loss:
            best_val_loss = valid_losses[-1]
            torch.save(model.state_dict(), best_model_path)
            print(f"□ Saved best model at {best_model_path} (Val Loss:
{best_val_loss:.4f})")

        print(f"Epoch [{epoch+1}/{num_epochs}] - Loss: {train_losses[-
1]:.4f} - Acc: {train_accuracies[-1]:.4f} - Val Loss: {valid_losses[-
1]:.4f} - Val Acc: {valid_accuracies[-1]:.4f}")

    return train_losses, valid_losses, train_accuracies,
valid_accuracies

def plot_loss_and_accuracy(train_losses, valid_losses,
train_accuracies, valid_accuracies):
    """
    Función para graficar la pérdida y precisión de entrenamiento y
validación a lo largo de las épocas.

    Args:
    - train_losses: Lista de pérdidas de entrenamiento para cada
época.
    - valid_losses: Lista de pérdidas de validación para cada época.
    - train_accuracies: Lista de precisiones de entrenamiento para
cada época.
    - valid_accuracies: Lista de precisiones de validación para cada
época.
    """
    plt.figure(figsize=(12, 5))

    # Plot Loss
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label="Train Loss")
```

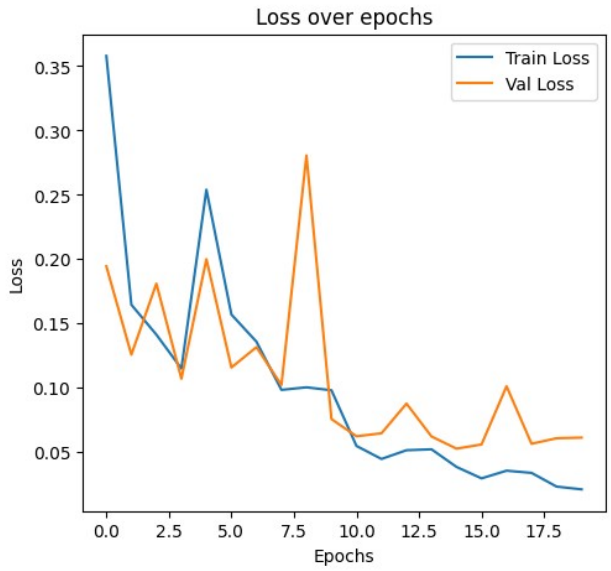
```
    plt.plot(valid_losses, label="Val Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Loss over epochs")
    plt.legend()

    # Plot Accuracy
    plt.subplot(1, 2, 2)
    plt.plot(train_accuracies, label="Train Acc")
    plt.plot(valid_accuracies, label="Val Acc")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.title("Accuracy over epochs")
    plt.legend()

    plt.show()

def evaluate_model(model, test_loader, device, best_model_path):
    """
    Carga un modelo y lo evalúa en el conjunto de prueba utilizando un
DataLoader para evitar problemas de memoria.

    Parámetros:
    - model: El modelo PyTorch entrenado.
    - test_loader: DataLoader para el conjunto de prueba.
    - device: El dispositivo (CPU/GPU) a usar para el cálculo.
    - best_model_path: Ruta para guardar el mejor modelo.
    """

    # Cargar el mejor modelo
    model.load_state_dict(torch.load(best_model_path))
    model.to(device)

    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            X_batch = X_batch.to(device)
            outputs = model(X_batch)
            preds = (outputs >= 0.5).float().cpu().numpy()
            all_preds.extend(preds)
            all_labels.extend(y_batch.cpu().numpy())

    # Calculate accuracy
    test_accuracy = accuracy_score(all_labels, all_preds)
    print(f"Precisión en el conjunto de prueba: {test_accuracy:.4f}")
    auc_roc = roc_auc_score(all_labels, all_preds)
    print(f"AUC-ROC en el conjunto de prueba: {auc_roc:.4f}")
```

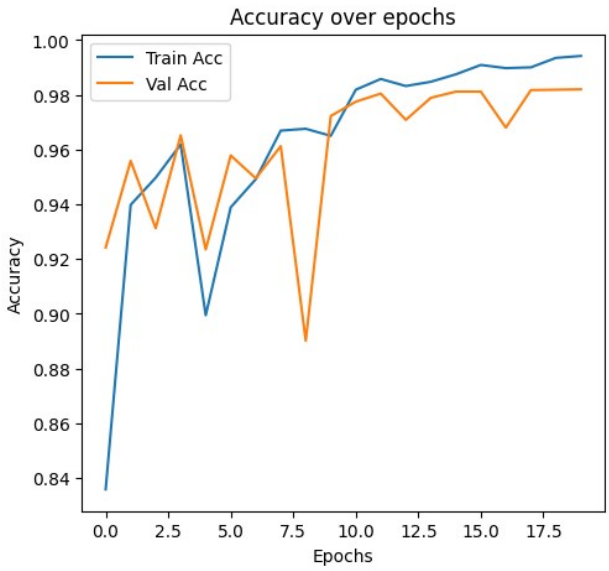
Entrenamiento y evaluación

```
# --------------------
# Entrenamiento y evaluación
# --------------------

# Instanciar modelo, pérdida y optimizador
model = LSTMModel().to(device)
criterion = nn.BCELoss()  # Pérdida para clasificación binaria
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Variables de ntrenamiento
num_epochs = 20
train_losses, valid_losses = [], []
train_accuracies, valid_accuracies = [], []
best_model_path = "../models/Test.pth"  # Ruta donde se guardará el
mejor modelo

# Entrenar el modelo
train_losses, valid_losses, train_accuracies, valid_accuracies =
train_model(
    model, train_loader, valid_loader, criterion, optimizer,
num_epochs, device, best_model_path
)

# Función para graficar la pérdida y precisión
plot_loss_and_accuracy(train_losses, valid_losses, train_accuracies,
valid_accuracies)

# Evaluar el modelo en el conjunto de prueba
evaluate_model(model, test_loader, device, best_model_path)
```

```
 Saved best model at ../models/Test.pth (Val Loss: 0.1945)
Epoch [1/20] - Loss: 0.3579 - Acc: 0.8358 - Val Loss: 0.1945 - Val
Acc: 0.9242
 Saved best model at ../models/Test.pth (Val Loss: 0.1259)
Epoch [2/20] - Loss: 0.1646 - Acc: 0.9398 - Val Loss: 0.1259 - Val
Acc: 0.9559
Epoch [3/20] - Loss: 0.1414 - Acc: 0.9497 - Val Loss: 0.1810 - Val
Acc: 0.9313
 Saved best model at ../models/Test.pth (Val Loss: 0.1071)
Epoch [4/20] - Loss: 0.1150 - Acc: 0.9619 - Val Loss: 0.1071 - Val
Acc: 0.9652
Epoch [5/20] - Loss: 0.2540 - Acc: 0.8994 - Val Loss: 0.2000 - Val
Acc: 0.9235
Epoch [6/20] - Loss: 0.1570 - Acc: 0.9389 - Val Loss: 0.1159 - Val
Acc: 0.9579
Epoch [7/20] - Loss: 0.1359 - Acc: 0.9492 - Val Loss: 0.1317 - Val
Acc: 0.9495
 Saved best model at ../models/Test.pth (Val Loss: 0.1023)
Epoch [8/20] - Loss: 0.0985 - Acc: 0.9669 - Val Loss: 0.1023 - Val
```

```
Acc: 0.9613
Epoch [9/20] - Loss: 0.1005 - Acc: 0.9676 - Val Loss: 0.2806 - Val
Acc: 0.8901
 Saved best model at ../models/Test.pth (Val Loss: 0.0758)
Epoch [10/20] - Loss: 0.0982 - Acc: 0.9650 - Val Loss: 0.0758 - Val
Acc: 0.9723
 Saved best model at ../models/Test.pth (Val Loss: 0.0624)
Epoch [11/20] - Loss: 0.0549 - Acc: 0.9819 - Val Loss: 0.0624 - Val
Acc: 0.9775
Epoch [12/20] - Loss: 0.0448 - Acc: 0.9858 - Val Loss: 0.0648 - Val
Acc: 0.9805
Epoch [13/20] - Loss: 0.0516 - Acc: 0.9832 - Val Loss: 0.0878 - Val
Acc: 0.9709
 Saved best model at ../models/Test.pth (Val Loss: 0.0623)
Epoch [14/20] - Loss: 0.0523 - Acc: 0.9848 - Val Loss: 0.0623 - Val
Acc: 0.9789
 Saved best model at ../models/Test.pth (Val Loss: 0.0528)
Epoch [15/20] - Loss: 0.0386 - Acc: 0.9875 - Val Loss: 0.0528 - Val
Acc: 0.9812
Epoch [16/20] - Loss: 0.0297 - Acc: 0.9909 - Val Loss: 0.0561 - Val
Acc: 0.9812
Epoch [17/20] - Loss: 0.0357 - Acc: 0.9898 - Val Loss: 0.1013 - Val
Acc: 0.9680
Epoch [18/20] - Loss: 0.0340 - Acc: 0.9901 - Val Loss: 0.0566 - Val
Acc: 0.9818
Epoch [19/20] - Loss: 0.0233 - Acc: 0.9935 - Val Loss: 0.0609 - Val
Acc: 0.9819
Epoch [20/20] - Loss: 0.0212 - Acc: 0.9942 - Val Loss: 0.0614 - Val
Acc: 0.9820
```





```
C:\Users\guigr\AppData\Local\Temp\ipykernel_3784\2726994642.py:13:
FutureWarning: You are using `torch.load` with `weights_only=False`
(the current default value), which uses the default pickle module
implicitly. It is possible to construct malicious pickle data which
will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  model.load_state_dict(torch.load(best_model_path))

Precisión en el conjunto de prueba: 0.9813
AUC-ROC en el conjunto de prueba: 0.9815
```

Aunque el modelo alcanza un **98%** de precisión en el conjunto de prueba, se concluye que no supera el rendimiento de modelos anteriores, a pesar de ser **notablemente más complejo** desde el punto de vista computacional. Sin embargo, aún quedan algunos experimentos adicionales por realizar, los cuales podrían contribuir a mejorar estos resultados en las siguientes etapas.

Experimentos Adicionales con la Arquitectura LSTM

Mejoras en Tokenización y Estructura BiLSTM

Como primer paso, realizaremos algunos ajustes en los parámetros de tokenización:

- **max_words** se incrementará de 5,000 a 10,000, lo que permitirá incluir un vocabulario más amplio. Consideramos que esto es relevante, ya que cada palabra podría aportar información valiosa para clasificar una noticia como real o falsa.
- Por otro lado, **max_len** se reducirá a 1024 para reducir la complejidad.
- Se probará con varios **batch_sizes** (128, 256...)

```python
# --------------------
# Tokenización y secuencias
# --------------------
# Disponibilidad de GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando device: {device}")
```

```
# Número máximo de palabras a considerar en el vocabulario
max_words = 10000
# Longitud máxima de las secuencias - La media de longitud de los
textos es 1656 aproximadamente
max_len = 1024

# Inicializar el tokenizador de TensorFlow con un vocabulario limitado
y un token para palabras 'Out-Of-Vocabulary'
tokenizer = Tokenizer(num_words=max_words, oov_token='<OOV>')
# Ajustar el tokenizador al texto de entrenamiento
tokenizer.fit_on_texts(X_train)

# Convertir textos a secuencias de índices y aplicar padding
X_train_embedding = tokenizer.texts_to_sequences(X_train)
X_train_embedding = pad_sequences(X_train_embedding, maxlen=max_len)
X_valid_embedding = tokenizer.texts_to_sequences(X_valid)
X_valid_embedding = pad_sequences(X_valid_embedding, maxlen=max_len)
X_test_embedding = tokenizer.texts_to_sequences(X_test)
X_test_embedding = pad_sequences(X_test_embedding, maxlen=max_len)

# Convertir a tensores de PyTorch
X_train_tensor = torch.tensor(X_train_embedding, dtype=torch.long)
y_train_tensor = torch.tensor(y_train.values,
dtype=torch.float32).unsqueeze(1)
X_valid_tensor = torch.tensor(X_valid_embedding, dtype=torch.long)
y_valid_tensor = torch.tensor(y_valid.values,
dtype=torch.float32).unsqueeze(1)
X_test_tensor = torch.tensor(X_test_embedding, dtype=torch.long)
y_test_tensor = torch.tensor(y_test.values,
dtype=torch.float32).unsqueeze(1)

print("Shape of X_train_tensor:", X_train_tensor.shape)
print("Shape of y_train_tensor:", y_train_tensor.shape)
print("Shape of X_valid_tensor:", X_valid_tensor.shape)
print("Shape of y_valid_tensor:", y_valid_tensor.shape)
print("Shape of X_test_tensor:", X_test_tensor.shape)
print("Shape of y_test_tensor:", y_test_tensor.shape)

# Batch Size
batch_size = 128

# Crear Datasets & DataLoaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
valid_dataset = TensorDataset(X_valid_tensor, y_valid_tensor)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size,
shuffle=False)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
```

```
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

Usando device: cuda
Shape of X_train_tensor: torch.Size([28283, 1024])
Shape of y_train_tensor: torch.Size([28283, 1])
Shape of X_valid_tensor: torch.Size([7071, 1024])
Shape of y_valid_tensor: torch.Size([7071, 1])
Shape of X_test_tensor: torch.Size([8839, 1024])
Shape of y_test_tensor: torch.Size([8839, 1])
```

La estructura del modelo LSTM anterior ha sido mejorada significativamente en esta nueva versión:

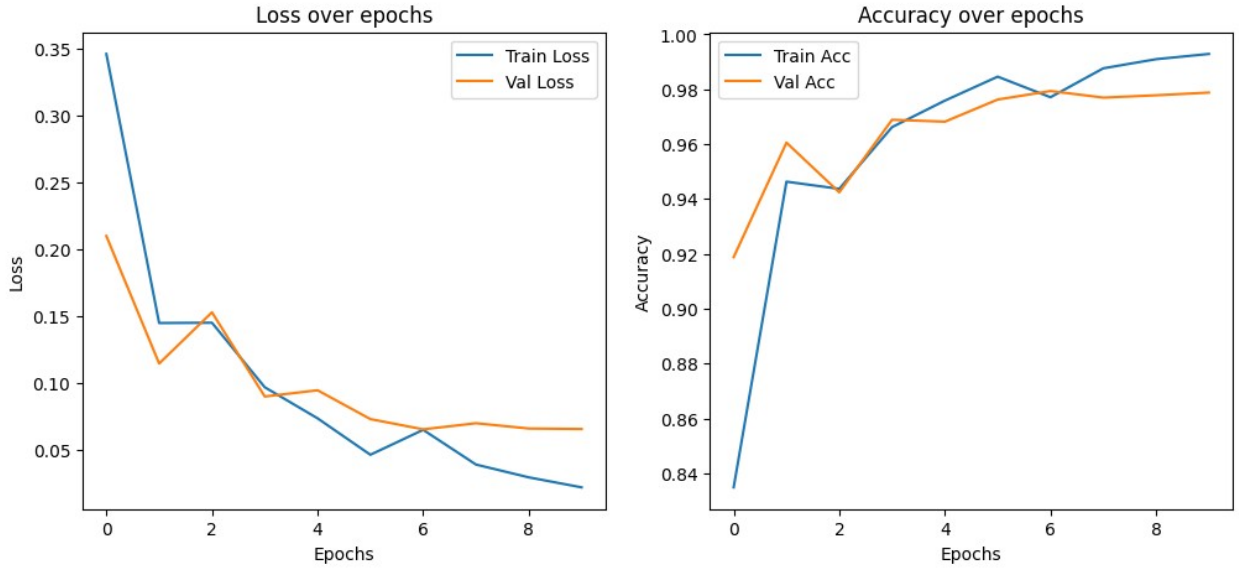
- La capa de **embedding** ahora tiene 64 dimensiones, lo que permite capturar representaciones semánticas más ricas de cada palabra.
- La LSTM simple se reemplaza por **dos capas BiLSTM** (bidireccionales), aportando mayor profundidad para modelar relaciones complejas y capturando contexto tanto hacia adelante como hacia atrás (pasado ⇄ futuro).
- La única capa lineal final se expande a dos capas densas ($64 \rightarrow 16 \rightarrow 1$), aumentando la capacidad del modelo para aprender y generalizar sobre patrones más abstractos.

Esta arquitectura busca una mejor comprensión del contexto completo de cada noticia que vayamos a clasificar como *verdadera* o *falsa*.

```
class BiLSTM(nn.Module):
    def __init__(self, input_dim=max_words, embedding_dim=64,
hidden_dim_1=64, hidden_dim_2=32):
        super(BiLSTM, self).__init__()

        self.embedding = nn.Embedding(
            num_embeddings=input_dim,
            embedding_dim=embedding_dim,
            padding_idx=0
        )

        self.bilstm1 = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim_1,
            batch_first=True,
            bidirectional=True
        )

        self.bilstm2 = nn.LSTM(
            input_size=hidden_dim_1*2,  # Porque es bidireccional (64
* 2)
            hidden_size=hidden_dim_2,
```

```
            batch_first=True,
            bidirectional=True
        )

        self.fc1 = nn.Linear(hidden_dim_2 * 2, 16)  # Salida de la
segunda BiLSTM
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(16, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)  # (batch_size, seq_len) → (batch_size,
seq_len, 64)
        out, _ = self.bilstm1(x)  # (batch_size, seq_len, 128)
        out, _ = self.bilstm2(out)  # (batch_size, seq_len, 64)
        # Tomamos el último paso de la secuencia
        out = out[:, -1, :]  # (batch_size, 64)
        out = self.fc1(out) # (batch_size, 16)
        out = self.relu(out)
        out = self.fc2(out) # (batch_size, 1)
        return self.sigmoid(out)  # Para clasificación binaria

# --------------------
# Entrenamiento y evaluación
# --------------------

# Instanciar modelo, pérdida y optimizador
model = BiLSTM().to(device)
criterion = nn.BCELoss()  # Pérdida para clasificación binaria
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Variables de entrenamiento
num_epochs = 10
train_losses, valid_losses = [], []
train_accuracies, valid_accuracies = [], []
best_model_path = "models/best_bilstm.pth"  # Ruta donde se guardará
el mejor modelo

# Entrenar el modelo
train_losses, valid_losses, train_accuracies, valid_accuracies =
train_model(
    model, train_loader, valid_loader, criterion, optimizer,
num_epochs, device, best_model_path
)

# Función para graficar la pérdida y precisión
plot_loss_and_accuracy(train_losses, valid_losses, train_accuracies,
valid_accuracies)
```

```
 Saved best model at models/Test.pth (Val Loss: 0.2096)
Epoch [1/10] - Loss: 0.3457 - Acc: 0.8350 - Val Loss: 0.2096 - Val
Acc: 0.9188
 Saved best model at models/Test.pth (Val Loss: 0.1143)
Epoch [2/10] - Loss: 0.1446 - Acc: 0.9463 - Val Loss: 0.1143 - Val
Acc: 0.9605
Epoch [3/10] - Loss: 0.1448 - Acc: 0.9437 - Val Loss: 0.1526 - Val
Acc: 0.9424
 Saved best model at models/Test.pth (Val Loss: 0.0896)
Epoch [4/10] - Loss: 0.0966 - Acc: 0.9662 - Val Loss: 0.0896 - Val
Acc: 0.9689
Epoch [5/10] - Loss: 0.0733 - Acc: 0.9759 - Val Loss: 0.0943 - Val
Acc: 0.9682
 Saved best model at models/Test.pth (Val Loss: 0.0727)
Epoch [6/10] - Loss: 0.0461 - Acc: 0.9846 - Val Loss: 0.0727 - Val
Acc: 0.9762
 Saved best model at models/Test.pth (Val Loss: 0.0651)
Epoch [7/10] - Loss: 0.0647 - Acc: 0.9771 - Val Loss: 0.0651 - Val
Acc: 0.9794
Epoch [8/10] - Loss: 0.0388 - Acc: 0.9876 - Val Loss: 0.0696 - Val
Acc: 0.9769
Epoch [9/10] - Loss: 0.0292 - Acc: 0.9909 - Val Loss: 0.0657 - Val
Acc: 0.9778
Epoch [10/10] - Loss: 0.0217 - Acc: 0.9929 - Val Loss: 0.0653 - Val
Acc: 0.9788
```




```
best_model_path = "models/best_bilstm.pth"
evaluate_model(model, test_loader, device, best_model_path)

C:\Users\guigr\AppData\Local\Temp\ipykernel_9120\2726994642.py:13:
FutureWarning: You are using `torch.load` with `weights_only=False`
```

```
(the current default value), which uses the default pickle module
implicitly. It is possible to construct malicious pickle data which
will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  model.load_state_dict(torch.load(best_model_path))

Precisión en el conjunto de prueba: 0.9888
AUC-ROC en el conjunto de prueba: 0.9889
```

Podemos observar un rendimiento sobresaliente, alcanzando una precisión cercana al **99%** en el conjunto de prueba.

Además, cabe destacar que se han realizado distintas pruebas modificando **diversos parámetros** y se ha llegado a las siguientes conclusiones:

- batch_size: Disminuirlo empeora el resultado, aumentarlo tampoco lo mejora y hace el entrenamiento más lento.
- max_len: Aumentarlo no mejora el resultado significativamente, por lo que se seguirá usando *max_len=1024*.
- num_epochs: A partir de la época 10 el modelo comienza a sobreajustar al conjunto de entrenamiento.

Vectorización mediante Embeddings Preentrenados (*Word2Vec*)

En este enfoque, utilizamos **Word2Vec** en lugar de la capa de embeddings tradicional de PyTorch. Word2Vec es una técnica de word embeddings que aprende representaciones vectoriales densas de palabras a partir de un gran corpus de texto, de forma que palabras con contextos similares tienen vectores similares. A diferencia de un tokenizador con embeddings entrenables dentro de una red neuronal, Word2Vec entrena un **modelo independiente** que luego usamos para **codificar nuestras secuencias de texto**.

Tokenizaremos el texto con la función *simple_preprocess*, y entrenamos un modelo Word2Vec sobre los datos tokenizados del conjunto de entrenamiento. Posteriormente, al igual que en el ejemplo anterior, definimos una función que convierte cada documento en una secuencia de vectores utilizando el modelo Word2Vec, aplicando padding para asegurar una longitud fija por documento. Finalmente, transformamos los conjuntos obteniendo así tensores de forma (n_samples, max_len=1024, vector_size=64), manteniendo las dimensiones del ejemplo anterior.

De esta manera, podremos comparar si ofrecen mejores resultados los embeddings entrenados junto con la red neuronal o aquellos obtenidos previamente a través de un modelo ya entrenado como Word2Vec.

```
# Cargar el DataFrame limpio
df = pd.read_csv("Datasets/Cleaned-FR-News_V2.csv")

# Dividimos los datos en entrenamiento y prueba
# Por ahora usaremos únicamente el texto de la noticia (omitimos el
título)
X = df["clean_text"]
y = df["label"]
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Se usará para redes neuronales
# Usaremos un 20% del conjunto de datos para validación (16% del
total)
X_train, X_valid, y_train, y_valid = train_test_split(X_train,
y_train, test_size=0.2, random_state=42)

# Tokenize the text data
print("Tokenizing text data...")
X_train_tokens = [simple_preprocess(text) for text in X_train]
X_valid_tokens = [simple_preprocess(text) for text in X_valid]
X_test_tokens = [simple_preprocess(text) for text in X_test]

# Train Word2Vec model on the training data
print("Training Word2Vec model...")
word2vec_model = Word2Vec(
    sentences=X_train_tokens,
    vector_size=64,    # Dimensionalidad de los embeddings
    window=10,         # Ventana para predecir el contexto
    min_count=5,       # Eliminar palabras raras (mínimo cada palabra
debe aparecer 5 veces en el corpus)
    workers=4,         # Usar 4 hilos de CPU
)

def encode_text_with_word2vec(tokens_list, model, max_len):
    """
    Convierte una lista de documentos tokenizados en secuencias de
embeddings de Word2Vec.

    Parámetros:
    - tokens_list: Lista de documentos tokenizados (lista de listas de
palabras).
    - model: Modelo Word2Vec entrenado.
    - max_len: Longitud máxima de la secuencia para el padding.

    Retorna:
```

```
    - Un array de NumPy con forma (num_samples, max_len, vector_size)
    """
    embedded_sequences = []

    for tokens in tokens_list:
        embeddings = [model.wv[word] for word in tokens if word in
model.wv]  # Convertir palabras a vectores
        embedded_sequences.append(embeddings)

    # Rellenar secuencias con ceros para garantizar una forma uniforme
    padded_embeddings = pad_sequences(embedded_sequences,
maxlen=max_len, dtype='float32', padding='pre', truncating='pre',
value=0.0)

    return np.array(padded_embeddings)

# Longitud máxima - Teniendo en cuenta que de media tenemos 1656
palabras.
max_len = 1024

# Convertir los datos de texto a embeddings de Word2Vec
print("Encoding train text data with Word2Vec...")
X_train_embedding = encode_text_with_word2vec(X_train_tokens,
word2vec_model, max_len)
print("Encoding valid text data with Word2Vec...")
X_valid_embedding = encode_text_with_word2vec(X_valid_tokens,
word2vec_model, max_len)
print("Encoding test text data with Word2Vec...")
X_test_embedding = encode_text_with_word2vec(X_test_tokens,
word2vec_model, max_len)

print("Shape of X_train:", X_train_embedding.shape)
print("Shape of X_valid:", X_valid_embedding.shape)
print("Shape of X_test:", X_test_embedding.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_valid:", y_valid.shape)
print("Shape of y_test:", y_test.shape)
```

```
Tokenizing text data...
Training Word2Vec model...
Encoding train text data with Word2Vec...
Encoding valid text data with Word2Vec...
Encoding test text data with Word2Vec...
Shape of X_train: (28283, 1024, 64)
Shape of X_valid: (7071, 1024, 64)
Shape of X_test: (8839, 1024, 64)
Shape of y_train: (28283,)
Shape of y_valid: (7071,)
Shape of y_test: (8839,)
```

```python
# GPU disponible
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
torch.cuda.empty_cache()
print("CUDA cache cleared.")

# Convertir datos a tensores de PyTorch cuando ya son arrays
def tensorize_w2v(data, labels):
    data_tensor = torch.tensor(data, dtype=torch.float32)
    labels_tensor = torch.tensor(labels.values,
dtype=torch.float32).unsqueeze(1)
    return data_tensor, labels_tensor

X_train_tensor, y_train_tensor = tensorize_w2v(X_train_embedding,
y_train)
X_valid_tensor, y_valid_tensor = tensorize_w2v(X_valid_embedding,
y_valid)
X_test_tensor, y_test_tensor = tensorize_w2v(X_test_embedding, y_test)

print("Shape of X_train_tensor:", X_train_tensor.shape)
print("Shape of y_train_tensor:", y_train_tensor.shape)
print("Shape of X_valid_tensor:", X_valid_tensor.shape)
print("Shape of y_valid_tensor:", y_valid_tensor.shape)
print("Shape of X_test_tensor:", X_test_tensor.shape)
print("Shape of y_test_tensor:", y_test_tensor.shape)

# Variables de configuración
batch_size = 128

# Create DataLoaders para entrenamiento y validación
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
valid_dataset = TensorDataset(X_valid_tensor, y_valid_tensor)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size,
shuffle=False)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)
```

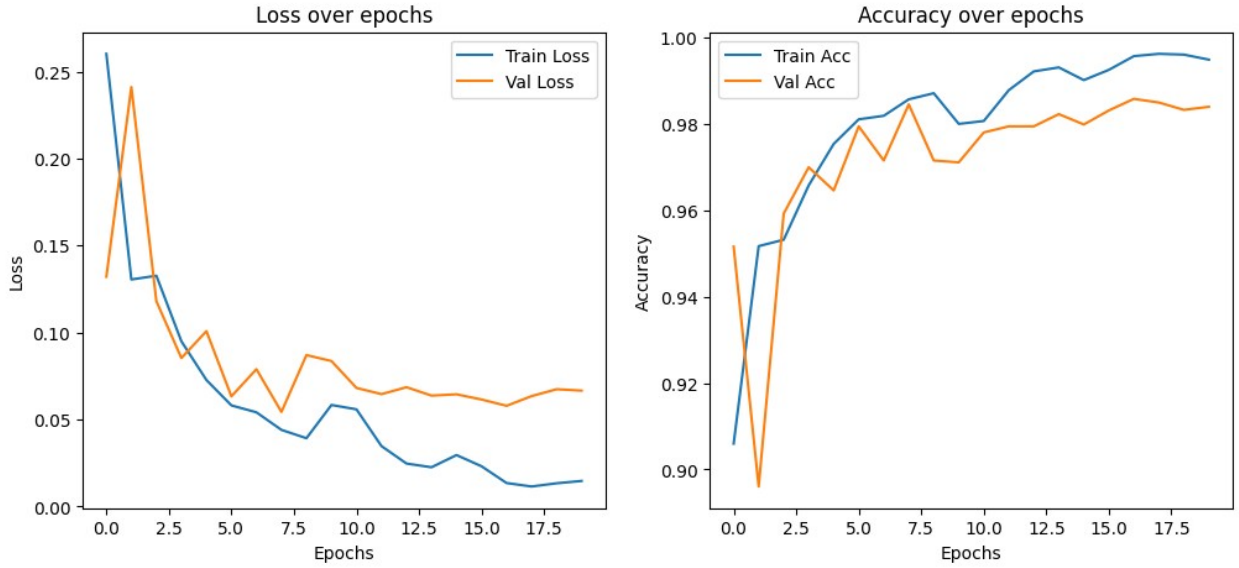
```
Using device: cuda
CUDA cache cleared.
Shape of X_train_tensor: torch.Size([28283, 1024, 64])
Shape of y_train_tensor: torch.Size([28283, 1])
Shape of X_valid_tensor: torch.Size([7071, 1024, 64])
Shape of y_valid_tensor: torch.Size([7071, 1])
Shape of X_test_tensor: torch.Size([8839, 1024, 64])
Shape of y_test_tensor: torch.Size([8839, 1])
```

```
class BiLSTM_W2V(nn.Module):
    def __init__(self, embedding_dim=64, hidden_dim_1=64,
hidden_dim_2=32):
        super(BiLSTM_W2V, self).__init__()

        # Eliminamos la capa de embeddings entrenables

        self.bilstm1 = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim_1,
            batch_first=True,
            bidirectional=True
        )

        self.bilstm2 = nn.LSTM(
            input_size=hidden_dim_1*2,  # Porque es bidireccional (64
* 2)
            hidden_size=hidden_dim_2,
            batch_first=True,
            bidirectional=True
        )

        self.fc1 = nn.Linear(hidden_dim_2 * 2, 16)  # Salida de la
segunda BiLSTM
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(16, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        out, _ = self.bilstm1(x)  # (batch_size, seq_len, 128)
        out, _ = self.bilstm2(out)  # (batch_size, seq_len, 64)
        # Tomamos el último paso de la secuencia
        out = out[:, -1, :]  # (batch_size, 64)
        out = self.fc1(out)
        out = self.relu(out)
        out = self.fc2(out)
        return self.sigmoid(out)  # Para clasificación binaria

# Instanciar modelo y moverlo a GPU si está disponible
model = BiLSTM_W2V().to(device)

# Función de pérdida y optimizador
criterion = nn.BCELoss()  # Pérdida para clasificación binaria
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Variables de entrenamiento
num_epochs = 20
train_losses, valid_losses = [], []
train_accuracies, valid_accuracies = [], []
```

```python
# Ruta donde se guardará el mejor modelo
best_model_path = "models/best_lstm_w2v.pth"

# Entrenar el modelo
train_losses, valid_losses, train_accuracies, valid_accuracies =
train_model(
    model, train_loader, valid_loader, criterion, optimizer,
num_epochs, device, best_model_path
)

# Función para graficar la pérdida y precisión
plot_loss_and_accuracy(train_losses, valid_losses, train_accuracies,
valid_accuracies)

# Evaluar el modelo en el conjunto de prueba
evaluate_model(model, test_loader, device, best_model_path)
```

```
 Saved best model at models/best_lstm_w2v.pth (Val Loss: 0.1321)
Epoch [1/20] - Loss: 0.2602 - Acc: 0.9060 - Val Loss: 0.1321 - Val
Acc: 0.9516
Epoch [2/20] - Loss: 0.1305 - Acc: 0.9517 - Val Loss: 0.2411 - Val
Acc: 0.8961
 Saved best model at models/best_lstm_w2v.pth (Val Loss: 0.1181)
Epoch [3/20] - Loss: 0.1327 - Acc: 0.9532 - Val Loss: 0.1181 - Val
Acc: 0.9593
 Saved best model at models/best_lstm_w2v.pth (Val Loss: 0.0854)
Epoch [4/20] - Loss: 0.0950 - Acc: 0.9658 - Val Loss: 0.0854 - Val
Acc: 0.9700
Epoch [5/20] - Loss: 0.0728 - Acc: 0.9754 - Val Loss: 0.1008 - Val
Acc: 0.9646
 Saved best model at models/best_lstm_w2v.pth (Val Loss: 0.0634)
Epoch [6/20] - Loss: 0.0582 - Acc: 0.9811 - Val Loss: 0.0634 - Val
Acc: 0.9795
Epoch [7/20] - Loss: 0.0542 - Acc: 0.9819 - Val Loss: 0.0790 - Val
Acc: 0.9716
 Saved best model at models/best_lstm_w2v.pth (Val Loss: 0.0544)
Epoch [8/20] - Loss: 0.0441 - Acc: 0.9858 - Val Loss: 0.0544 - Val
Acc: 0.9846
Epoch [9/20] - Loss: 0.0393 - Acc: 0.9872 - Val Loss: 0.0871 - Val
Acc: 0.9716
Epoch [10/20] - Loss: 0.0584 - Acc: 0.9801 - Val Loss: 0.0837 - Val
Acc: 0.9711
Epoch [11/20] - Loss: 0.0559 - Acc: 0.9807 - Val Loss: 0.0682 - Val
Acc: 0.9781
Epoch [12/20] - Loss: 0.0348 - Acc: 0.9879 - Val Loss: 0.0646 - Val
Acc: 0.9795
Epoch [13/20] - Loss: 0.0247 - Acc: 0.9922 - Val Loss: 0.0686 - Val
Acc: 0.9795
Epoch [14/20] - Loss: 0.0226 - Acc: 0.9931 - Val Loss: 0.0638 - Val
Acc: 0.9823
```

```
Epoch [15/20] - Loss: 0.0296 - Acc: 0.9902 - Val Loss: 0.0645 - Val
Acc: 0.9799
Epoch [16/20] - Loss: 0.0232 - Acc: 0.9926 - Val Loss: 0.0616 - Val
Acc: 0.9832
Epoch [17/20] - Loss: 0.0136 - Acc: 0.9958 - Val Loss: 0.0579 - Val
Acc: 0.9859
Epoch [18/20] - Loss: 0.0116 - Acc: 0.9963 - Val Loss: 0.0635 - Val
Acc: 0.9850
Epoch [19/20] - Loss: 0.0134 - Acc: 0.9961 - Val Loss: 0.0675 - Val
Acc: 0.9833
Epoch [20/20] - Loss: 0.0147 - Acc: 0.9949 - Val Loss: 0.0667 - Val
Acc: 0.9840
```




```
C:\Users\guigr\AppData\Local\Temp\ipykernel_17928\2726994642.py:13:
FutureWarning: You are using `torch.load` with `weights_only=False`
(the current default value), which uses the default pickle module
implicitly. It is possible to construct malicious pickle data which
will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  model.load_state_dict(torch.load(best_model_path))
```

```
Precisión en el conjunto de prueba: 0.9834
AUC-ROC en el conjunto de prueba: 0.9835
```

De nuevo, obtenemos un gran resultado con una precisión en el conjunto de prueba del **98.34%**, aunque no se supera el rendimiento obtenido utilizando el *Tokenizer* junto con la capa de *Embeddings entrenables* durante la red neuronal.

¿Por qué la vectorización mediante Embeddings como *BERT* o *allMiniLM* (token a token) no tiene sentido?

1. Consumiría demasiada memoria: Generar embeddings de **768** dimensiones para cada token en secuencias de **512** para los aproximadamente **28.000** textos únicamente del conjunto de train crearía **tensores gigantescos (40 GB)** que no caben en la RAM.
2. Es **computacionalmente carísimo**, procesar estas secuencias masivas con una LSTM sería **extremadamente lento**.
3. Probablemente es **redundante**, haciendo que la LSTM sobre todos los tokens sea potencialmente innecesaria.
4. Beneficio marginal vs. coste: Con un 99% de precisión ya alcanzado, **la posible mejora no justifica el inmenso coste y complejidad**.

Introducción de una capa de Self-Attention

```python
# --------------------
# Tokenización y secuencias
# --------------------
# Disponibilidad de GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando device: {device}")

# Número máximo de palabras a considerar en el vocabulario
max_words = 10000
# Longitud máxima de las secuencias - La media de longitud de los
textos es 1656 aproximadamente
max_len = 1024

# Inicializar el tokenizador de TensorFlow con un vocabulario limitado
y un token para palabras 'Out-Of-Vocabulary'
tokenizer = Tokenizer(num_words=max_words, oov_token='<OOV>')
# Ajustar el tokenizador al texto de entrenamiento
tokenizer.fit_on_texts(X_train)

# Convertir textos a secuencias de índices y aplicar padding
X_train_embedding = tokenizer.texts_to_sequences(X_train)
X_train_embedding = pad_sequences(X_train_embedding, maxlen=max_len)
X_valid_embedding = tokenizer.texts_to_sequences(X_valid)
X_valid_embedding = pad_sequences(X_valid_embedding, maxlen=max_len)
X_test_embedding = tokenizer.texts_to_sequences(X_test)
X_test_embedding = pad_sequences(X_test_embedding, maxlen=max_len)

# Convertir a tensores de PyTorch
```

```
X_train_tensor = torch.tensor(X_train_embedding, dtype=torch.long)
y_train_tensor = torch.tensor(y_train.values,
dtype=torch.float32).unsqueeze(1)
X_valid_tensor = torch.tensor(X_valid_embedding, dtype=torch.long)
y_valid_tensor = torch.tensor(y_valid.values,
dtype=torch.float32).unsqueeze(1)
X_test_tensor = torch.tensor(X_test_embedding, dtype=torch.long)
y_test_tensor = torch.tensor(y_test.values,
dtype=torch.float32).unsqueeze(1)

print("Shape of X_train_tensor:", X_train_tensor.shape)
print("Shape of y_train_tensor:", y_train_tensor.shape)
print("Shape of X_valid_tensor:", X_valid_tensor.shape)
print("Shape of y_valid_tensor:", y_valid_tensor.shape)
print("Shape of X_test_tensor:", X_test_tensor.shape)
print("Shape of y_test_tensor:", y_test_tensor.shape)

# Batch Size
batch_size = 128

# Crear Datasets & DataLoaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
valid_dataset = TensorDataset(X_valid_tensor, y_valid_tensor)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size,
shuffle=False)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

Usando device: cuda
Shape of X_train_tensor: torch.Size([28283, 1024])
Shape of y_train_tensor: torch.Size([28283, 1])
Shape of X_valid_tensor: torch.Size([7071, 1024])
Shape of y_valid_tensor: torch.Size([7071, 1])
Shape of X_test_tensor: torch.Size([8839, 1024])
Shape of y_test_tensor: torch.Size([8839, 1])
```

Anteriormente, el modelo BiLSTM procesaba toda la noticia pero, para decidir si era real o falsa, se basaba principalmente en la información resumida en **el último estado oculto** (el resultado de out[:, -1, :]). La idea era que este último estado ya contenía una buena síntesis de todo lo leído. Sin embargo, esta estrategia fija le da más peso al final del texto y podría **pasar por alto pistas cruciales que aparezcan al principio o en medio de la noticia**.

Aunque existen alternativas como **Max Pooling** o **Mean Pooling**, se optó por incorporar una capa de **Self-Attention**. Al colocar la Self-Attention después de la BiLSTM, en lugar de utilizar únicamente el último estado oculto, la atención examina toda la secuencia de estados ocultos que generó la BiLSTM. Así, **aprende dinámicamente a asignar pesos de importancia a cada parte de la secuencia** (a cada palabra o frase representada por su estado oculto). De esta

manera, crea un nuevo vector resumen **(vector de contexto)** que es una **combinación ponderada de toda la información**, dando más énfasis a las partes que el propio modelo considera más relevantes para la clasificación, sin importar si están al principio, en medio o al final.

```
class BiLSTMAttention(nn.Module):
    def __init__(self, input_dim=max_words, embedding_dim=64,
hidden_dim_1=64, hidden_dim_2=32, num_heads=4):
        super(BiLSTMAttention, self).__init__()

        self.embedding = nn.Embedding(
            num_embeddings=input_dim,
            embedding_dim=embedding_dim,
            padding_idx=0
        )

        self.bilstm1 = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim_1,
            batch_first=True,
            bidirectional=True
        )

        self.bilstm2 = nn.LSTM(
            input_size=hidden_dim_1 * 2,
            hidden_size=hidden_dim_2,
            batch_first=True,
            bidirectional=True
        )

        bilstm_output_dim = hidden_dim_2 * 2

        if bilstm_output_dim % num_heads != 0:
             raise ValueError(f"hidden_dim_2 * 2 ({bilstm_output_dim})
debe ser divisible por num_heads ({num_heads})")
        self.attention = nn.MultiheadAttention(
            embed_dim=bilstm_output_dim, # Dimensión total de
entrada/salida
            num_heads=num_heads,         # Número de cabezas de
atención
            dropout=0.1,                 # Dropout opcional en
atención
            batch_first=True             # Para que acepte (batch,
seq, feature)
        )

        self.fc1 = nn.Linear(bilstm_output_dim, 16) # La dimensión de
salida de la atención es la misma que la entrada
        self.relu = nn.ReLU()
```

```
        self.fc2 = nn.Linear(16, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # x shape: (batch_size, seq_len)
        x = self.embedding(x)  # (batch_size, seq_len, embedding_dim)
        out, _ = self.bilstm1(x)  # (batch_size, seq_len, hidden_dim_1
* 2)
        out, _ = self.bilstm2(out) # (batch_size, seq_len,
hidden_dim_2 * 2)

        # MultiheadAttention espera query, key, value.
        # Devuelve: attn_output (batch, seq, embed_dim),
attn_output_weights (batch, num_heads, seq, seq) o similar
        attn_output, attn_weights = self.attention(out, out, out) #
attn_output shape: (batch_size, seq_len, hidden_dim_2 * 2)

        # Promediamos los outputs de atención a lo largo de la
secuencia.
        context = torch.mean(attn_output, dim=1) # (batch_size,
hidden_dim_2 * 2)
        # ------------------------

        # Usamos el vector de contexto en lugar del último estado
oculto
        out = self.fc1(context) # (batch_size, 16)
        out = self.relu(out)
        out = self.fc2(out)     # (batch_size, 1)
        return self.sigmoid(out)
```

Debido a la introducción de la capa de Self-Attention, que es muy costosa computacionalmente, vamos a introducir **Early-Stopping** en el método de entrenamiento para poder pararlo cuando la pérdida haya dejado de mejorar.

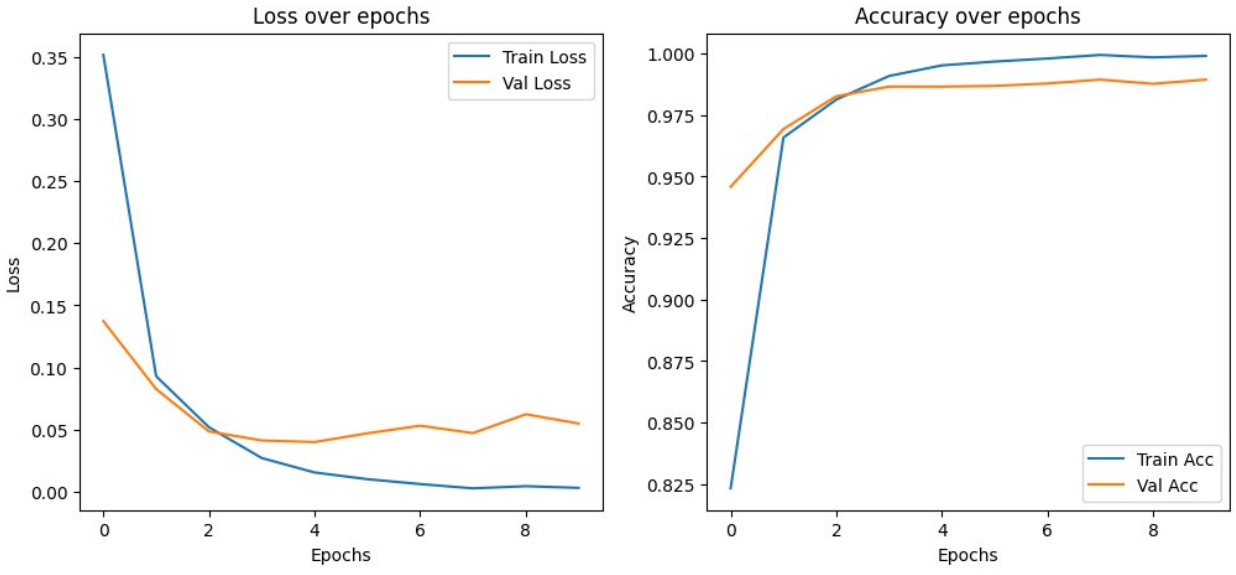
```
def train_model_optimized(
    model, train_loader, valid_loader, criterion, optimizer,
    num_epochs, device, best_model_path, patience=5
):
    train_losses, valid_losses = [], []
    train_accuracies, valid_accuracies = [], []
    best_val_loss = float("inf")
    epochs_without_improvement = 0

    for epoch in range(num_epochs):
        model.train()
        train_loss, correct_train, total_train = 0, 0, 0

        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
```

```
            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            predicted = (outputs >= 0.5).float()
            correct_train += (predicted == y_batch).sum().item()
            total_train += y_batch.size(0)

        train_losses.append(train_loss / len(train_loader))
        train_accuracies.append(correct_train / total_train)

        # Validation
        model.eval()
        valid_loss, correct_valid, total_valid = 0, 0, 0

        with torch.no_grad():
            for X_batch, y_batch in valid_loader:
                X_batch, y_batch = X_batch.to(device),
y_batch.to(device)
                outputs = model(X_batch)
                loss = criterion(outputs, y_batch)
                valid_loss += loss.item()
                predicted = (outputs >= 0.5).float()
                correct_valid += (predicted == y_batch).sum().item()
                total_valid += y_batch.size(0)

        val_loss_avg = valid_loss / len(valid_loader)
        val_acc = correct_valid / total_valid
        valid_losses.append(val_loss_avg)
        valid_accuracies.append(val_acc)

        # Early stopping
        if val_loss_avg < best_val_loss:
            best_val_loss = val_loss_avg
            torch.save(model.state_dict(), best_model_path)
            print(f"□ Saved best model at {best_model_path} (Val Loss:
{best_val_loss:.4f})")
            epochs_without_improvement = 0
        else:
            epochs_without_improvement += 1
            print(f"□ No improvement for {epochs_without_improvement}
epoch(s)")

        print(f"Epoch [{epoch+1}/{num_epochs}] - Loss: {train_losses[-
1]:.4f} - Acc: {train_accuracies[-1]:.4f} - Val Loss:
{val_loss_avg:.4f} - Val Acc: {val_acc:.4f}")
```

```
        if epochs_without_improvement >= patience:
            print(f"⏹ Early stopping triggered after {epoch+1}
epochs.")
            break

    return train_losses, valid_losses, train_accuracies,
valid_accuracies

# Limpiar la caché de CUDA para liberar memoria
torch.cuda.empty_cache()
print("CUDA cache cleared.")

# Instanciar modelo y moverlo a GPU si está disponible
model = BiLSTMAttention().to(device)

# Función de pérdida y optimizador
criterion = nn.BCELoss()  # Pérdida para clasificación binaria
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Variables de entrenamiento
num_epochs = 20
train_losses, valid_losses = [], []
train_accuracies, valid_accuracies = [], []

# Ruta donde se guardará el mejor modelo
best_model_path = "models/best_lstm_attention.pth"

# Entrenar el modelo
train_losses, valid_losses, train_accuracies, valid_accuracies =
train_model_optimized(
    model, train_loader, valid_loader, criterion, optimizer,
num_epochs, device, best_model_path, patience=5
)

# Función para graficar la pérdida y precisión
plot_loss_and_accuracy(train_losses, valid_losses, train_accuracies,
valid_accuracies)

# Evaluar el modelo en el conjunto de prueba
evaluate_model(model, test_loader, device, best_model_path)
```

```
CUDA cache cleared.
⏹ Saved best model at models/best_lstm_attention.pth (Val Loss:
0.1374)
Epoch [1/20] - Loss: 0.3516 - Acc: 0.8233 - Val Loss: 0.1374 - Val
Acc: 0.9458
⏹ Saved best model at models/best_lstm_attention.pth (Val Loss:
0.0827)
Epoch [2/20] - Loss: 0.0930 - Acc: 0.9659 - Val Loss: 0.0827 - Val
Acc: 0.9693
```

```
 Saved best model at models/best_lstm_attention.pth (Val Loss:
0.0486)
Epoch [3/20] - Loss: 0.0519 - Acc: 0.9813 - Val Loss: 0.0486 - Val
Acc: 0.9826
 Saved best model at models/best_lstm_attention.pth (Val Loss:
0.0413)
Epoch [4/20] - Loss: 0.0272 - Acc: 0.9909 - Val Loss: 0.0413 - Val
Acc: 0.9866
 Saved best model at models/best_lstm_attention.pth (Val Loss:
0.0400)
Epoch [5/20] - Loss: 0.0156 - Acc: 0.9952 - Val Loss: 0.0400 - Val
Acc: 0.9866
 No improvement for 1 epoch(s)
Epoch [6/20] - Loss: 0.0102 - Acc: 0.9967 - Val Loss: 0.0471 - Val
Acc: 0.9868
 No improvement for 2 epoch(s)
Epoch [7/20] - Loss: 0.0062 - Acc: 0.9980 - Val Loss: 0.0532 - Val
Acc: 0.9878
 No improvement for 3 epoch(s)
Epoch [8/20] - Loss: 0.0028 - Acc: 0.9994 - Val Loss: 0.0472 - Val
Acc: 0.9894
 No improvement for 4 epoch(s)
Epoch [9/20] - Loss: 0.0045 - Acc: 0.9984 - Val Loss: 0.0624 - Val
Acc: 0.9877
 No improvement for 5 epoch(s)
Epoch [10/20] - Loss: 0.0032 - Acc: 0.9990 - Val Loss: 0.0550 - Val
Acc: 0.9894
 Early stopping triggered after 10 epochs.
```




```
C:\Users\guigr\AppData\Local\Temp\ipykernel_10880\2726994642.py:13:
FutureWarning: You are using `torch.load` with `weights_only=False`
```

```
(the current default value), which uses the default pickle module
implicitly. It is possible to construct malicious pickle data which
will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  model.load_state_dict(torch.load(best_model_path))

Precisión en el conjunto de prueba: 0.9888
AUC-ROC en el conjunto de prueba: 0.9888
```

Una vez más, logramos una **precisión del 99%** en el conjunto de prueba. Sin embargo, el coste computacional asociado con la incorporación de la capa de atención (el entrenamiento duró alrededor de 50 minutos, en contraste con el otro modelo que alcanzaba el mismo rendimiento en solo 2 minutos) no justifica la diferencia. De hecho, destacaría que se alcanza **exactamente la misma precisión** que con el modelo anterior, aunque en este caso la **convergencia a la pérdida óptima ocurre mucho más rápidamente**, alcanzándose en la quinta época (0.04).

Tras todos estos experimentos, llegamos a la conclusión de que, para el split que estamos utilizando, la precisión es **difícilmente mejorable**.

¿Vale la pena optimizar hiperparámetros?

Se concluyó que, dado que el mejor modelo ya alcanza un 99% de precisión, no resulta justificado emplear frameworks de **optimización de hiperparámetros** como *Optuna* en este caso. El entrenamiento más reciente tomó cerca de 60 minutos, y el coste computacional —tanto en tiempo como en recursos— de ejecutar múltiples pruebas variando parámetros como *lstm_units* o *batch_size* (los cuales ya fueron explorados previamente en experimentos reducidos) sería **elevado en relación con la posible mejora**, que como máximo sería del 1%.

¿Y aumentar **max_words**?

Vamos a realizar un análisis sobre cómo el Tokenizer transforma las oraciones y cuántas palabras terminan siendo codificadas en el proceso.

```
# Decodificar la primera secuencia de prueba y mostrar ejemplo con
tokens <OOV>
decoded_texts = tokenizer.sequences_to_texts([X_test_embedding[0]])
num_non_oov_words = sum(1 for word in decoded_texts[0].split() if word
!= '<OOV>')
print(f"Ejemplo de oración con tokens <OOV>: {decoded_texts[0]}")
print(f"Número de palabras en el texto decodificado:
{len(decoded_texts[0].split())}, Número de palabras que no son <OOV>:
```

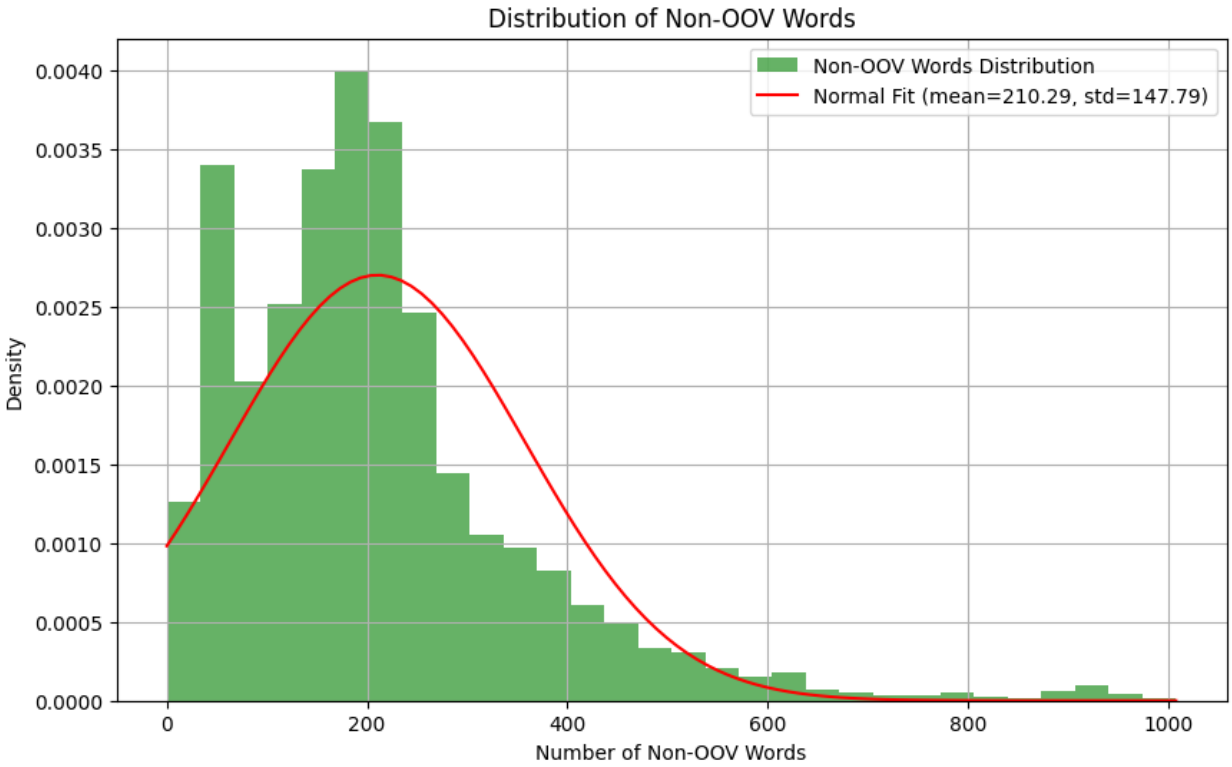
```
{num_non_oov_words}")

# Buscar el X_test_embedding con el mínimo y máximo número de palabras
<OOV>
min_oov_index = None
max_oov_index = None
min_oov_count = float('inf')
max_oov_count = float('-inf')

for i, embedding in enumerate(X_test_embedding):
    decoded_text = tokenizer.sequences_to_texts([embedding])[0]
    oov_count = sum(1 for word in decoded_text.split() if word ==
'<OOV>')

    if oov_count < min_oov_count:
        min_oov_count = oov_count
        min_oov_index = i

    if oov_count > max_oov_count:
        max_oov_count = oov_count
        max_oov_index = i

print()
print(f"Índice con el mínimo número de palabras <OOV>:
{min_oov_index}, Cantidad: {min_oov_count}")
print(f"Índice con el máximo número de palabras <OOV>:
{max_oov_index}, Cantidad: {max_oov_count}")

# Contar el número de embeddings con más de 1000, más de 500 y menos
de 100 palabras que no son <OOV>
count_more_than_1000 = 0
count_more_than_500 = 0
count_less_than_100 = 0

for embedding in X_test_embedding:
    decoded_text = tokenizer.sequences_to_texts([embedding])[0]
    non_oov_count = sum(1 for word in decoded_text.split() if word !=
'<OOV>')

    if non_oov_count > 1000:
        count_more_than_1000 += 1
    if non_oov_count > 500:
        count_more_than_500 += 1
    if non_oov_count < 100:
        count_less_than_100 += 1

print()
print(f"Número de embeddings: {len(X_test_embedding)}")
print(f"Número de embeddings con más de 1000 palabras que no son
<OOV>: {count_more_than_1000}")
```

```
print(f"Número de embeddings con más de 500 palabras que no son <OOV>:
{count_more_than_500}")
print(f"Número de embeddings con menos de 100 palabras que no son
<OOV>: {count_less_than_100}")

# Calcular el número de palabras que no son <OOV> para cada embedding
non_oov_counts = [
    sum(1 for word in tokenizer.sequences_to_texts([embedding])
[0].split() if word != '<OOV>')
    for embedding in X_test_embedding
]

# Graficar la distribución
plt.figure(figsize=(10, 6))
plt.hist(non_oov_counts, bins=30, density=True, alpha=0.6, color='g',
label='Distribución de palabras no <OOV>')

# Ajustar una distribución normal
mean, std = np.mean(non_oov_counts), np.std(non_oov_counts)
x = np.linspace(min(non_oov_counts), max(non_oov_counts), 100)
pdf = (1 / (std * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x - mean) /
std) ** 2)
plt.plot(x, pdf, 'r', label=f'Ajuste normal (media={mean:.2f},
std={std:.2f})')

plt.title('Distribución de palabras que no son <OOV>')
plt.xlabel('Número de palabras que no son <OOV>')
plt.ylabel('Densidad')
plt.legend()
plt.grid()
plt.show()
```

```
Example of sentence with <OOV> tokens: <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
```

```
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
<00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V> <00V>
```

```
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
<OOV> <OOV> <OOV> <OOV> <OOV> warsaw poland low house parliament
friday approve change country national judiciary council part broad
overhaul court system push rule conservative new rule still need
senate presidential approval parliament responsibility choose member
council turn nominate judge
Number of words in the decoded text: 1024, Number of non-<OOV> words:
35

Index with minimum <OOV> words: 4265, Count: 16
Index with maximum <OOV> words: 4983, Count: 1024

Number of embeddings: 8839
Number of embeddings with more than 1000 non-<OOV> words: 2
Number of embeddings with more than 500 non-<OOV> words: 405
Number of embeddings with less than 100 non-<OOV> words: 1964
```



Al imprimir cualquier ejemplo de una frase procesada por el *Tokenizer*, observamos que la mayoría de las frases contienen en su mayoría el token 'OOV'. Por ejemplo, en la anterior frase, solo 35 de los 1024 tokens corresponden a palabras reales.

De hecho, al analizar la **distribución del número de palabras Non-OOV** (Out-of-Vocabulary) en cada frase de 1024 tokens, encontramos que la media está cerca de 200. Es decir, lo común es que alrededor del 80% del texto codificado sea el token 'OOV', el cual no aporta información útil.

Aunque esto podría sugerir que sería necesario aumentar el tamaño del vocabulario, dado que el Tokenizer selecciona las palabras más frecuentes (las cuales tienden a ser las más relevantes, como vimos con TF-IDF), se realizaron pruebas duplicando el valor de max_words a 20.000, pero los resultados no mejoraron. De hecho, en algunos casos, podría ser más beneficioso reducir ese valor para simplificar la complejidad.

Comprobando resultados de Test

A continuación, se verifica que un total de 99 textos fueron clasificados incorrectamente, de un total de 8839 pertenecientes al conjunto de prueba. Cabe destacar que estos textos son normales, es decir, no presentan transformaciones u otras condiciones que puedan justificar una mala clasificación.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando device: {device}")
model = BiLSTM().to(device)
```

```
best_model_path = "models/best_bilstm.pth"
evaluate_model(model, test_loader, device, best_model_path)
```

```
Usando device: cuda
```

```
C:\Users\guigr\AppData\Local\Temp\ipykernel_18592\2726994642.py:13:
FutureWarning: You are using `torch.load` with `weights_only=False`
(the current default value), which uses the default pickle module
implicitly. It is possible to construct malicious pickle data which
will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  model.load_state_dict(torch.load(best_model_path))
```

```
Precisión en el conjunto de prueba: 0.9888
AUC-ROC en el conjunto de prueba: 0.9889
```

```
# Evaluar el modelo en el conjunto de prueba y obtener predicciones
model.load_state_dict(torch.load(best_model_path))
model.to(device)
model.eval()

misclassified_indices = []
misclassified_texts = []
misclassified_labels = []
misclassified_predictions = []

with torch.no_grad():
    for i, (X_batch, y_batch) in enumerate(test_loader):
        X_batch = X_batch.to(device)
        y_batch = y_batch.cpu().numpy()
        outputs = model(X_batch).cpu().numpy()
        preds = (outputs >= 0.5).astype(int)

        # Identificar índices donde las predicciones no coinciden con
las etiquetas reales
        for j in range(len(y_batch)):
            if preds[j] != y_batch[j]:
                misclassified_indices.append(i * batch_size + j)
                misclassified_texts.append(X_test.iloc[i * batch_size
+ j])
                misclassified_labels.append(y_batch[j])
```

```
                misclassified_predictions.append(preds[j])

# Crear un DataFrame con las noticias mal clasificadas
misclassified_df = pd.DataFrame({
    "Text": misclassified_texts,
    "True Label": misclassified_labels,
    "Predicted Label": misclassified_predictions
})

print(f"Total misclassified samples: {len(misclassified_df)}")
misclassified_df.head()
```

```
C:\Users\guigr\AppData\Local\Temp\ipykernel_3544\255346102.py:2:
FutureWarning: You are using `torch.load` with `weights_only=False`
(the current default value), which uses the default pickle module
implicitly. It is possible to construct malicious pickle data which
will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  model.load_state_dict(torch.load(best_model_path))

Total misclassified samples: 99
```

	Text	True Label	Predicted Label
0	los angeles comedian kathy griffin tearfully ...	[1.0]	[0]
1	also map show state accept refugee syria make ...	[0.0]	[1]
2	turn trump rally oppose republican frontrunner...	[0.0]	[1]
3	davos switzerland global economy well shape ye...	[1.0]	[0]
4	president trump announce play golf quickly jup...	[0.0]	[1]