BETO (el BERT para español)

es mucho más poderoso que FastText porque entiende contexto, emojis, frases cortas y lenguaje natural moderno. Perfecto para tu dataset de mensajes cortos y con símbolos raros.

Flujo con BETO + MLP

1. Tokenizar mensajes con el tokenizer de BETO.
2. Obtener embeddings: usar la salida de BETO (ej. last_hidden_state.mean(dim=1) → vector de cada mensaje).
3. Entrenar MLP con esos embeddings.
4. Clasificar spam (1) vs legit (0).

In [25]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report, confusion_matrix
import torch
from transformers import AutoTokenizer, AutoModel
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam, RMSprop, SGD, Adagrad
from tensorflow.keras.callbacks import EarlyStopping
#from transformers import logging
#logging.set_verbosity_error()
```

In [2]:
```
# ===== 1. Cargar datos =====
df = pd.read_excel("datasetv2.xlsx")
```

In [3]:
```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2025 NVIDIA Corporation
Built on Wed_Apr__9_19:29:17_Pacific_Daylight_Time_2025
Cuda compilation tools, release 12.9, V12.9.41
Build cuda_12.9.r12.9/compiler.35813241_0
```

In [4]:
```
#!pip3 install torch torchvision --index-url https://download.pytorch.org/whl/cu
#!pip install transformers
#!pip install ipywidgets
#!pip install hf_xet
```

In [4]:
```
# ===== 2. Cargar BETO =====
tokenizer = AutoTokenizer.from_pretrained("dccuchile/bert-base-spanish-wwm-uncas
beto = AutoModel.from_pretrained("dccuchile/bert-base-spanish-wwm-uncased")
```

```
Some weights of BertModel were not initialized from the model checkpoint at dccuc
hile/bert-base-spanish-wwm-uncased and are newly initialized: ['pooler.dense.bia
s', 'pooler.dense.weight']
You should probably TRAIN this model on a down-stream task to be able to use it f
or predictions and inference.
```

In [5]:
```python
# ===== 3. Generar embeddings de los mensajes =====
def mensaje_a_embedding(texto, tokenizer, model, max_len=64):
    inputs = tokenizer(texto, return_tensors="pt", truncation=True, padding="max
    with torch.no_grad():
        outputs = model(**inputs)
    # Usamos el promedio de las representaciones de las palabras
    embedding = outputs.last_hidden_state.mean(dim=1).squeeze().numpy()
    return embedding
```

In [6]:
```python
X = np.vstack([mensaje_a_embedding(msg, tokenizer, beto) for msg in df["message"
```

In [8]:
```python
# ===== 4. Etiquetas =====
encoder = LabelEncoder()
y = encoder.fit_transform(df["target"])  # spam=1, legit=0
```

In [9]:
```python
# ===== 5. Train/Test Split =====
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
```

In [17]:
```python
# ===== 6. Modelo MLP =====
model = Sequential([
    Dense(256, activation="relu"),
    Dropout(0.4),
    Dense(128, activation="relu"),
    Dropout(0.3),
    Dense(1, activation="sigmoid")
])
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

In [18]:
```python
#agregamos EarlyStopping
early_stop = EarlyStopping(
    monitor='val_loss',       # Métrica a observar
    patience=5,               # Cuántas epochs esperar sin mejora
    restore_best_weights=True  # Recuperar los mejores pesos
)
```

In [19]:
```python
# ===== 7. Entrenamiento =====
history = model.fit(X_train, y_train,
                    validation_data=(X_test, y_test),
                    epochs=100, batch_size=16,callbacks=[early_stop])
```

```
Epoch 1/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9081 - loss: 0.2427 - val_
accuracy: 0.9462 - val_loss: 0.1456
Epoch 2/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9425 - loss: 0.1596 - val_
accuracy: 0.9563 - val_loss: 0.1290
Epoch 3/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9500 - loss: 0.1393 - val_
accuracy: 0.9592 - val_loss: 0.1206
Epoch 4/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9528 - loss: 0.1265 - val_
accuracy: 0.9580 - val_loss: 0.1193
Epoch 5/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9559 - loss: 0.1177 - val_
accuracy: 0.9563 - val_loss: 0.1175
Epoch 6/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9582 - loss: 0.1072 - val_
accuracy: 0.9563 - val_loss: 0.1134
Epoch 7/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9619 - loss: 0.1002 - val_
accuracy: 0.9586 - val_loss: 0.1118
Epoch 8/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9636 - loss: 0.0933 - val_
accuracy: 0.9569 - val_loss: 0.1130
Epoch 9/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9675 - loss: 0.0862 - val_
accuracy: 0.9563 - val_loss: 0.1117
Epoch 10/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9706 - loss: 0.0792 - val_
accuracy: 0.9569 - val_loss: 0.1109
Epoch 11/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9722 - loss: 0.0721 - val_
accuracy: 0.9574 - val_loss: 0.1149
Epoch 12/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9737 - loss: 0.0688 - val_
accuracy: 0.9616 - val_loss: 0.1121
Epoch 13/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9769 - loss: 0.0617 - val_
accuracy: 0.9586 - val_loss: 0.1157
Epoch 14/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9805 - loss: 0.0540 - val_
accuracy: 0.9557 - val_loss: 0.1190
Epoch 15/100
423/423 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9814 - loss: 0.0516 - val_
accuracy: 0.9610 - val_loss: 0.1193
```

In [20]:
```python
# ===== 8. Evaluación =====
loss, acc = model.evaluate(X_test, y_test)
print(f"Accuracy con BETO: {acc:.4f}")
```

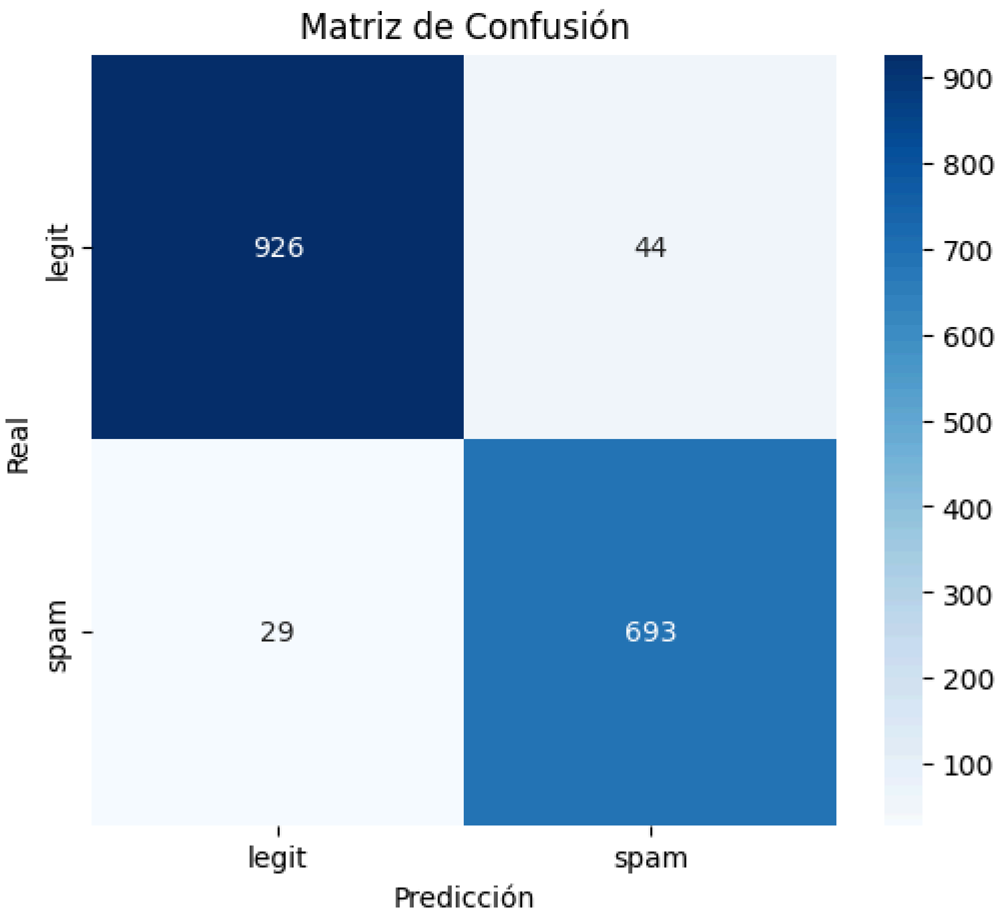
```
53/53 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9569 - loss: 0.1109
Accuracy con BETO: 0.9569
```

In [23]:
```python
y_pred = (model.predict(X_test) > 0.5).astype("int32")

print("\nClassification Report:\n", classification_report(y_test, y_pred, target
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
53/53 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step

Classification Report:
               precision    recall  f1-score   support

       legit       0.97      0.95      0.96       970
        spam       0.94      0.96      0.95       722

    accuracy                           0.96      1692
   macro avg       0.95      0.96      0.96      1692
weighted avg       0.96      0.96      0.96      1692


Confusion Matrix:
 [[926  44]
 [ 29 693]]
```

In [26]:

```python
# =======================
# 9. Matriz de confusión
# =======================
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["legit", "spam"]
plt.title("Matriz de Confusión")
plt.xlabel("Predicción")
plt.ylabel("Real")
plt.show()
```



In [27]:

```python
# =======================
# 8. Gráfico de entrenamiento
# =======================
plt.figure(figsize=(12,5))
```

```python
# Accuracy
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Entrenamiento')
plt.plot(history.history['val_accuracy'], label='Validación')
plt.title('Accuracy por Epoch')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Loss
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Entrenamiento')
plt.plot(history.history['val_loss'], label='Validación')
plt.title('Loss por Epoch')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```