

INTELIGENCIA ARTIFICIAL APLICADA A INTERNET DE LAS COSAS

Identificación de noticias falsas

ABSTRACT

La presente memoria aborda el desarrollo de un proyecto de Inteligencia Artificial para la clasificación de noticias en falsas o verdaderas.

Diego Pellicer y Alejandro de Celis
19 de mayo de 2025.

Índice

Glosario de términos.....	2
1. Introducción.....	3
2. División de tareas.....	5
3. Elección del dataset y el proyecto a realizar.....	6
4. Análisis del dataset y Preparación del dataset.....	7
5. Aplicación y evaluación de modelos IA.....	13
Modelo RNN - LSTM.....	13
RNN-LSTM, K-Fold Cross-Validation, K = 5.....	23
Modelo RNN - GRU.....	24
RNN-GRU, K-Fold Cross-Validation, K = 5.....	27
Modelo BERT.....	28
Modelos no PLN.....	37
Random Forest.....	38
Gradient Boosted Decision Tree.....	39
Linear SVM.....	41
Perceptrón multicapa.....	42
Conclusiones para los modelos IA estudiados.....	45
6. Aplicación de IA explicativa.....	49
LSTM - SHAP y LIME.....	51
SHAP:.....	51
LIME:.....	53
GRU - SHAP y LIME.....	54
SHAP:.....	54
Lime:.....	55
Transformers.....	56
7. Conclusiones.....	58

Glosario de términos

A continuación, incluimos un listado de siglas y acrónimos utilizados en el presente documento y su correspondiente significado.

- GPU: Graphics Processing Unit
- GRU: Gated Recurrent Unit
- IA: Inteligencia Artificial
- KFold: K-Fold Cross Validation
- LIME: Local Interpretable Model-Agnostic Explanations
- NLP: Natural Language Processing
- PLN: Procesamiento de Lenguaje Natural
- RNN: Recurrent Neural Network
- SHAP: SHapley Additive Explanations
- SVC: Support Vector Machine
- SLTM: Long Short-Term Memory
- SVM: Support Vector Machine
- TN: True Negative
- TP: True Positive
- FN: False Negative
- FP: False Positive
- URL: Uniform Resource Locator
- XAI:Explainable AI

1. Introducción

El presente trabajo tiene como objetivo el desarrollo de un sistema de identificación de potenciales noticias, por escrito, falsas. Se corresponde al trabajo final de la asignatura Inteligencia Artificial Aplicada a Internet de las Cosas, del máster Internet de las Cosas, de la Universidad Complutense de Madrid.

El equipo inicialmente estaba formado por tres alumnos, pero uno de ellos, Pablo Alcalde, causó baja, y el trabajo lo han realizado Diego Pellicer Lafuente y Alejandro de Celis Domínguez.

En este trabajo hemos aplicado los conocimientos adquiridos con la asignatura para:

1. Seleccionar un conjunto de datos o dataset de calidad y que nos permita hacer un trabajo de clasificación preciso, dentro de los límites propios de la tecnología a usar.
2. Estudio de varios modelos de Inteligencia Artificial, agrupados en Machine Learning y Redes Neuronales, para elegir aquel que mejor se adapte al objetivo del trabajo.
3. Desarrollo de sistemas de clasificación adaptados a la problemática a resolver con nuestro trabajo.

Hemos decidido trabajar principalmente con redes neuronales recurrentes (RNN), versiones LSTM y GRU, y transformers (BERT), de cara a complementar lo aprendido durante en las prácticas de la asignatura. Adicionalmente, hemos decidido complementar nuestro estudio con los siguientes modelos de IA:

- Random Forest
- Gradient Boosted Decision Tree
- Linear SVM
- Perceptrón multicapa

Para evaluar estos modelos, hemos utilizado:

- Matriz de dispersión. Hemos decidido mostrar la matriz tal y como la obtenemos y “pintando” la matriz.
- Classification report.

Para el desarrollo de código hemos utilizado el Github siguiente:

<https://github.com/Pelli223/aiiotFakeNews>

Este sitio tiene los siguientes ficheros:

- IA-TrabajoFinal-RNN-GRU-v1.ipynb. Código para RNN-GRU. Incluye validación simple y KFold, K=5.
- IA-TrabajoFinal-RNN-LSTM-v1.ipynb. Código para RNN-LSTM. Incluye validación simple y KFold, K=5.
- SupervisedModelsNoPLN.ipynb. Código de los modelos Supervisados.
- Transformers.ipynb. Código del modelo Transformers.

- analisisYLimpiezaDatos.ipynb. Código para la limpieza y estandarización del dataset.
- datos_limpios.csv. Versión limpia del dataset sin estandarizar.
- datos_limpios_estandarizados.csv. Versión en CSV usado para Transformers y SupervizedModelsNoPLN.
- datos_limpios_estandarizados.xlsx. Versión en Excel usado para RNN-GRU y LSTM.
- fake_news_dataset.csv. Primer dataset considerado. Descartado.
- 24-25-IAIC-TrabajoFinal-DiegoPellicer-AlejandrodeCelis-Memoria-v1.pdf. Memoria del trabajo.

2. División de tareas

A continuación, indicamos la división de tareas realizadas por los dos integrantes del grupo.

Tareas	Diego Pellicer	Alejandro de Celis
Identificación dataset inicial		✓
Creación Github y Google Drive	✓	✓
Actualización dataset	✓	
Preparación dataset	✓	
RNN-SLTM, división simple y KFold		✓
RNN-GRU, división simple y KFold		✓
BERT	✓	
Randomforest	✓	
Gradient Boosted Decision Tree	✓	
Linear SVM	✓	✓
Perceptrón multicapa		✓
XAI - RNN - SHAP		✓
XAI - RNN - Lime		✓
BERT - SHAP	✓	
Memoria	✓	✓

3. Elección del dataset y el proyecto a realizar

Inicialmente, elegimos este dataset:

<https://www.kaggle.com/datasets/khushikyad001/fake-news-detection>. Sin embargo, lo descartamos porque el problema que nos hemos encontrado es que en el texto de las noticias sólo tiene información irrelevante, por ejemplo: noticia 1, noticia 2. Creemos que es importante también utilizar la información disponible en el cuerpo de la noticia.

Tras investigar y analizar diversas opciones, hemos considerado usar el dataset siguiente: <https://www.kaggle.com/datasets/mahdimashayekhi/fake-news-detection-dataset>. Como podemos leer en la descripción del dataset, éste contiene 20 000 noticias generadas para simular las noticias que podemos encontrar. Se incluyen filas con campos vacíos para representar la dificultad de obtener toda la información asociada a una noticia.

Como indicamos en el punto anterior, el objetivo principal del trabajo ha sido la resolución de un problema de Inteligencia Artificial, en concreto, la identificación de potenciales noticias falsas, clasificándolas como ‘Verdadera’ o ‘Falsa’.

4. Análisis del dataset y Preparación del dataset

Para poder empezar a usar los datos de nuestro dataset en el entrenamiento de modelos para clasificación, primero debemos de analizar y preparar estos para su uso. En este dataset contamos con 20000 registros, lo primero que debemos de hacer es comprobar que ninguno de estos contengan valores nulos y en caso de contenerlos, eliminar esas filas ya que pueden ser problemáticas.

```
El df originalmente tenia 20000 columnas.  
El dataframe sin duplicados ni na tiene 18045 columnas.
```

El dataset, contaba con una serie de valores nulos, al contar con una gran cantidad de registros y siendo que estos no se reducen drásticamente al eliminar aquellas filas con valores nulos, decidimos eliminar estas para continuar con nuestro análisis. Además de preocuparnos por valores nulos, debemos de comprobar también la existencia de valores duplicados los cuales no existían para nuestro caso. Por último y en vistas de que nuestros sistemas de clasificación no tienden a clasificar hacia una de las dos posibles clases, se comprueba que el número de registros de ambas clases sea parejo, lo cual es el caso.

```
label
fake    9095
real    8950
Name: count, dtype: int64
```

Una vez realizado el análisis sobre los datos previos, el siguiente paso es analizar las clases, comprobar sus tipos y además decidir si todas son necesarias. Contamos con las siguientes clases en el dataframe:

- **title:** Se trata de una columna de texto la cual guarda el nombre de las noticias. Esta es necesaria para el entrenamiento de los modelos, además se deberá de usar técnicas de PLN para que esta pueda ser procesada por los modelos de aprendizaje usados.
- **text:** Contiene el contenido de la noticia, al igual que **title** se trata de una columna de texto que tendrá que ser preprocesada con técnicas de PLN.
- **date:** Columna que informa la fecha de la noticia en formato de texto, esta puede contener información relevante ya que las fake news pueden darse más a menudo en fechas determinadas como elecciones, ciertas festividades etc. Por esto no consideramos dejarla fuera del dataset, al menos a primera vista, pese a que puede no ser relevante en nuestro conjunto de datos concreto. Se requiere transformar esta a valores numéricos de día, mes y año para poder usarla en nuestros modelos.
- **source:** Se trata de del medio que emite la noticia, este es un dato categórico, el cual ha de ser tratado mediante one-hot encoding. Esto debido a que otra codificación como label encoding puede llegar a hacer pensar al modelo que hay distancias entre los distintos medios cosa que no es cierta y puede llevar a errores de clasificación.

- **author:** Refleja en texto el nombre del autor que hizo la noticia, se puede decir que es un dato categórico con un gran número de categorías. Es por esto que hemos decidido prescindir de él, ya que nos podemos topar con autores que no han aparecido en el entrenamiento o casos fuera del dataset que al igual no comparten autor. Por tanto pensamos que es un dato que aporta una categoría no muy útil para la clasificación de noticias debido a la inmensa cantidad de posibles valores que puede llegar a tomar.
- **category:** Como bien indica su nombre, es un valor categórico que nos guarda la información de la categoría a la que pertenece la noticia. Al igual que pasa con el campo **source** este ha de ser codificado con one-hot encoding ya que no existen distintas diferencias entre las categorías y por tanto hay que evitar crear distintas distancias entre ellas a la hora de entrenar los modelos.
- **label:** Se trata de la columna de clasificación de nuestro dataset. Esta nos dice si la noticia es falsa o no. Debido a que solo puede tomar dos valores y en vistas de no aumentar significativamente el número de columnas de nuestro dataset, esta columna categórica será codificada mediante label encoding.

Una vez analizadas las columnas de nuestro dataset, se procede a codificar y eliminar aquellas columnas que no consideramos necesarias.

```
# Eliminamos las columnas que consideramos innecesarias del dataframe
df.drop(columns=['author'], inplace=True)
```

```
df_encoded = pd.get_dummies(df, columns=['category'], dtype=int)
df_encoded = pd.get_dummies(df_encoded, columns=['source'], dtype=int)
df_encoded['label'] = df_encoded['label'].map({'fake': 0, 'real': 1})
```

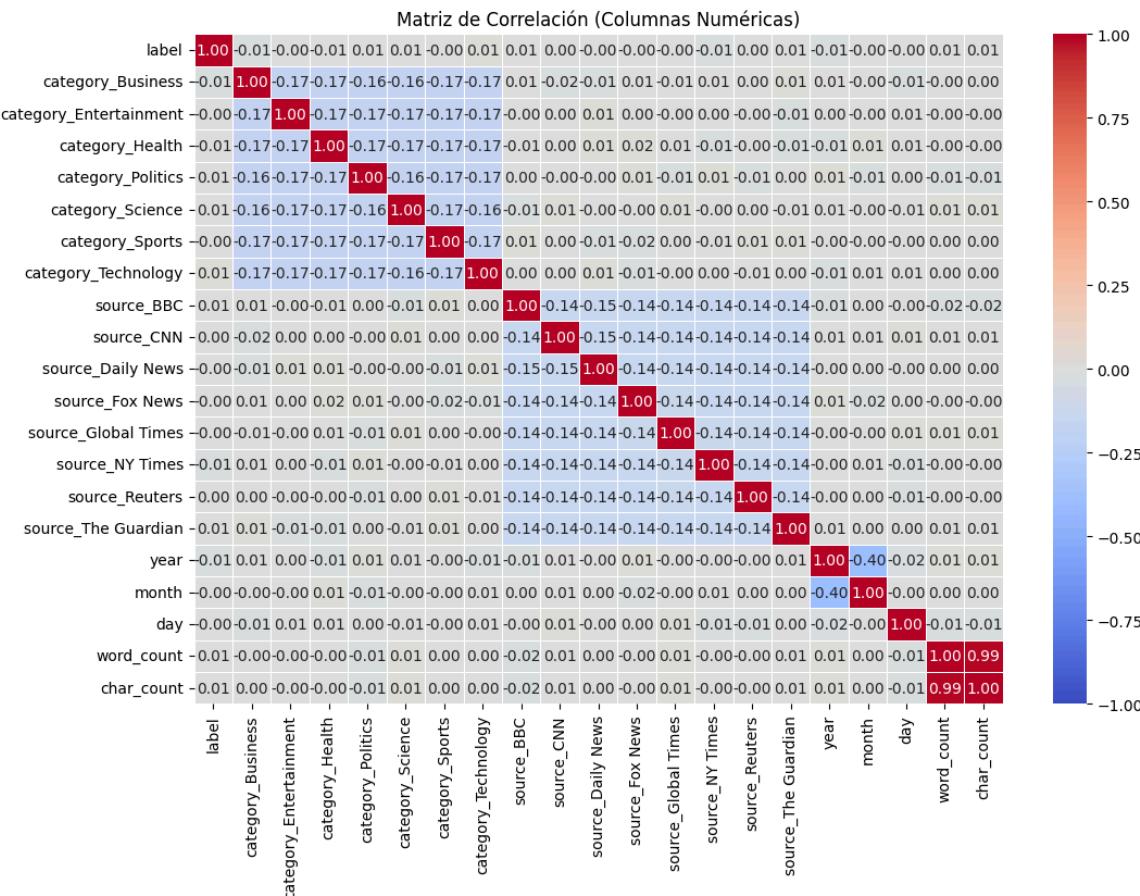
```
df_encoded["date"] = pd.to_datetime(df["date"], format="mixed", dayfirst=True)
df_encoded["year"] = df_encoded["date"].dt.year
df_encoded["month"] = df_encoded["date"].dt.month
df_encoded["day"] = df_encoded["date"].dt.day

df_encoded.drop(columns=['date'], inplace=True)
```

Con esto hemos conseguido que nuestro dataset, a excepción de los campos de texto, se encuentra ya listo para ser usado para el entrenamiento de modelos de machine learning. Sin embargo, consideramos que puede ser útil la generación de dos nuevos campos, uno que cuente el número de palabras del texto y otro que haga lo mismo con los caracteres. Esto en vista de que estos datos pueden contener información relevante que diferencie unas noticias de otras.

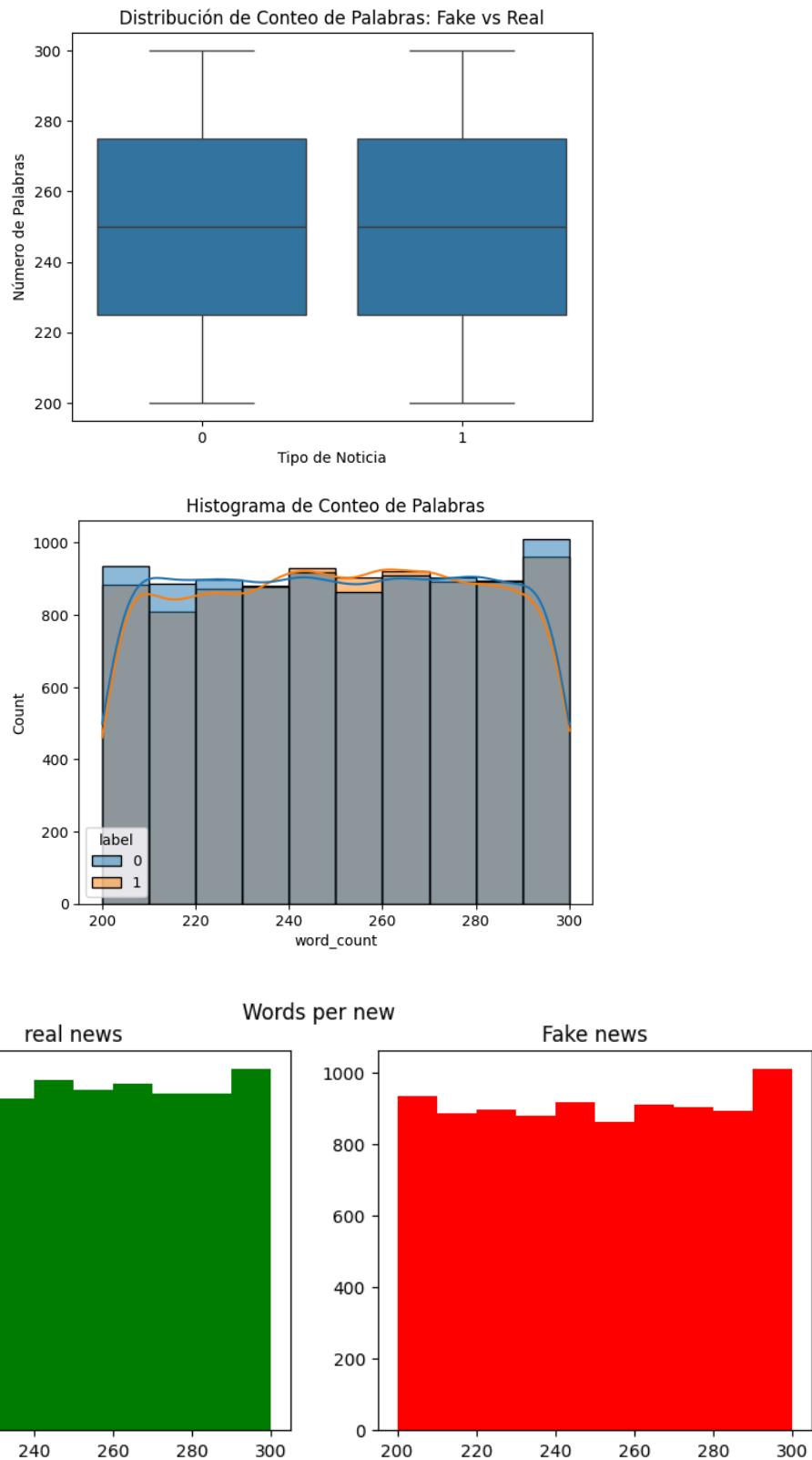
```
df_encoded['word_count'] = df_encoded['text'].apply(lambda x: len(x.split()))
df_encoded['char_count'] = df_encoded['text'].str.len()
```

Una vez tenemos todas nuestras columnas, el siguiente de los pasos es, para aquellas columnas que se encuentran listas para su uso en modelos, ver la correlación entre todas estas. A simple vista, las columnas originales del dataset, no parecen tener una correlación muy grande entre ellas y esto nos lo confirma la matriz de correlaciones realizada, que es el método usado para comprobar las correlaciones de los datos seleccionados.

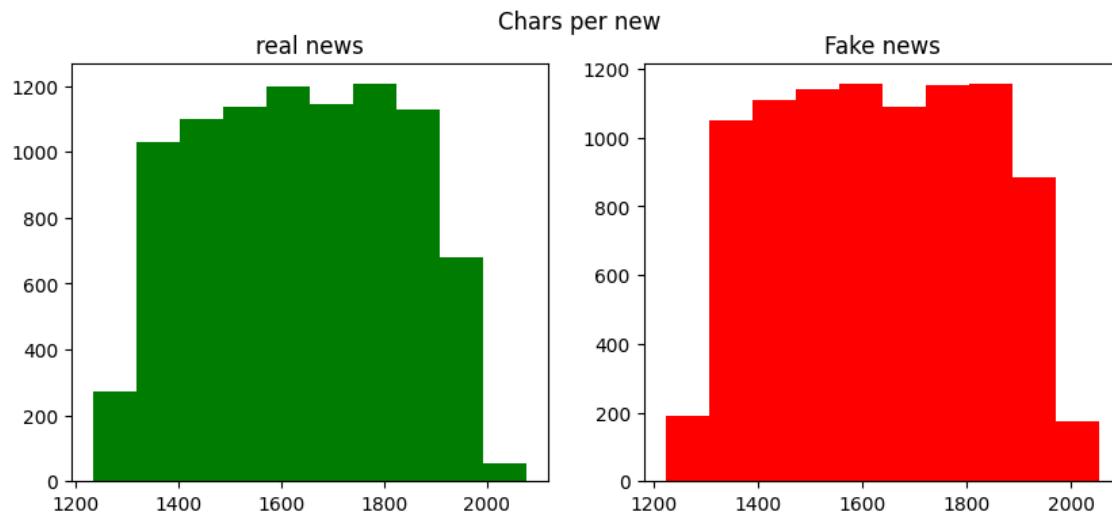


En la matriz podemos observar como apenas están correlacionadas las distintas columnas, se aprecia cierta correlación en aquellas que han sido codificadas pero ningún valor alto de correlación apreciable. Los únicos valores que se encuentran fuertemente correlacionados, son el número de caracteres y de palabras, esto tiene sentido, sin embargo era importante contar con ambos hasta esta comprobación ya que ciertas noticias podían contener múltiples signos de puntuación, exclamación etc. Y esto podría haber sido una información valiosa si ese tipo de patrones se diera más en un tipo de noticias que en otras. Sin embargo, al no ser el caso, se decide eliminar una de las dos columnas, la cual será el número de caracteres ya que si ambas representan prácticamente lo mismo tiene más sentido contar las palabras de las noticias.

Una vez comprobada las correlaciones entre las clases solo queda hacer un último análisis el cual consideramos interesante. Este se trata de comprobar si existe algún tipo de relación entre el número de palabras o caracteres y el tipo de noticia. Para ello se realizan dos gráficas para cada uno de los valores a comprobar, obteniendo para el número de palabras el siguiente resultado.



En estas gráficas se observa como los valores para las distintas clases son bastante semejantes, no llegando a haber una diferencia significativa entre las clases. Se puede llegar a ver como las noticias reales tienden a tener algo más de palabras pero nada destacable. A continuación, se hace la misma comparación con el número de caracteres.



En este caso los resultados son semejantes, sin embargo para este caso, son las noticias falsas las que tienden a tener algo más de caracteres pero al igual que con las palabras, la diferencia no es significativa.

Una vez tenemos todos los datos que serán usados para los entrenamientos, solo queda preparar las columnas de texto en vista de simplificar el texto de estas. Para ello, se hace uso de nlk para eliminar lo que se denominan “stopwords” en el texto. Estas se tratan de signos de puntuación, palabras como determinantes o proposiciones, las cuales no aportan un gran significado al texto y pueden crear ruido a la hora de entrenar los modelos. Además también se eliminan símbolos extraños que puedan contener los texto y que puedan afectar a los modelos de machine learning.

```

def limpiar_texto(texto):
    texto = texto.lower()
    texto = re.sub(r"><.*?>", "", texto) # quitar etiquetas HTML
    texto = re.sub(r"^[^a-zA-Záéíóúñü\s]", "", texto) # quitar puntuación y números
    texto = re.sub(r"\s+", " ", texto).strip()

    # Eliminar stopwords inglesas
    stop_words = set(stopwords.words('english'))
    texto = ' '.join([word for word in texto.split() if word not in stop_words])
    return texto

df_encoded["text"] = df_encoded["text"].apply(limpiar_texto)
df_encoded["title"] = df_encoded["title"].apply(limpiar_texto)

```

Una vez eliminadas las stopwords y los posibles símbolos raros, se procede a realizar un proceso conocido como Lemmatice, el cual consiste en reducir las palabras que encontramos en el texto a su forma original o su forma más básica. Esto se puede traducir en palabras como scientifics, ser reducidas a scientist etc.

```

# Apply lemmatization on tokens
def lemmatize(text):
    word_net = WordNetLemmatizer()
    return [word_net.lemmatize(word) for word in text]

df_encoded['text'] = df_encoded['text'].apply(lambda x: lemmatize(x.split()))
df_encoded['title'] = df_encoded['title'].apply(lambda x: lemmatize(x.split()))

```

Este paso, nos convierte los campos de texto en listas de palabras, por tanto debemos de volver a unir estas palabras para conservar todo como un solo texto.

```
df_encoded['text'] = df_encoded['text'].apply(lambda x: ' '.join(x))
df_encoded['title'] = df_encoded['title'].apply(lambda x: ' '.join(x))
```

Una vez realizado esto, ya tenemos nuestros campos de texto listos para más adelante utilizar distintos métodos de PLN para poder pasar el texto a algo que los modelos puedan entender.

Por último, vamos a generar dos nuevos datasets, uno de ellos contendrá todos aquellos datos de columnas numéricas estandarizados, el otro quedará con estos sin estandarizar. Esto en vista de ver si a algunos modelos la estandarización de los datos le pueda llegar a afectar de forma negativa.

```
df_encoded.to_csv('datos_limpios.csv', index=False)

scaler = StandardScaler()

df_normalized = df_encoded.drop(columns=['label'])

numeric_cols = df_normalized.select_dtypes(include=["int64", "float64", "int32"]).columns.tolist()

df_normalized[numeric_cols] = scaler.fit_transform(df_normalized[numeric_cols])
df_normalized['label'] = df_encoded['label']

df_normalized.to_csv('datos_limpios_estandarizados.csv', index=False)
```

Por último cabe mencionar que no se realiza en este paso un tratamiento concreto de los campos de texto para que estos estén listos para nuestros modelos debido a que este tratamiento depende del tipo de modelo a aplicar, como se verá en el siguiente capítulo.

5. Aplicación y evaluación de modelos IA

Para este trabajo utilizamos las redes neuronales profundas, que son redes neuronales con una gran cantidad de capas profundas. El motivo de esta elección es que nos encontramos con un dataset con una gran cantidad de datos no estructurados, como es el texto. Además también hacemos uso de transformers, los cuales son modelos de IA de PLN que utilizan técnicas de deep learning para trabajar con datos de manera no secuencial. Esta arquitectura permite a las máquinas entender y procesar el lenguaje humano de manera más eficiente y efectiva que otros modelos de machine learning.

Hemos desarrollado y evaluado un varios modelos perceptrón multicapa, en concreto:

- RNN. Red Neuronal Recurrente, usadas para procesar datos secuenciales, como textos. Se crean conexiones recurrentes, bucles, y como entrada de cada neurona, se usa la salida de la neurona anterior y las salidas de neuronas de capas previas, creándose así una memoria interna.
- Hemos elegido las dos variantes siguientes:
 - Memoria a corto-largo plazo (LSTM): Permite guardar información “más antigua” que lo conseguido con una red RNN estándar.
 - Unidades recurrentes cerradas (GRU): Mismo objetivo que redes LSTM: guardar información a largo plazo. Sin embargo, tiene una estructura más simple que LSTM y con menos puertas. Esto repercute en una red neuronal más fácil de entrenar y computacionalmente más eficiente.
- Transformers. Es una red neuronal basada en Deep Learning con un gran rendimiento en tareas de procesamiento de lenguaje natural y visión por computador. A diferencia de las otras redes neuronales que hemos usado, ésta no utiliza convolución o recurrencia y se basa en el concepto de mecanismo de atención, esto significa que se le da más importancia a partes específicas de los datos usados (por ejemplo, palabras en un texto) que nos puedan interesar para obtener una predicción. Esto nos ayuda en la eficiencia a la hora de generar predicciones. Los transformers son útiles ya que estos son capaces de guardar información como el contexto al tratar texto, para esta clasificación hemos decidido optar por el uso del modelo BERT de transformers para el tratamiento de texto en inglés.
- Modelos de machine learning no específicos para PLN. Hemos hecho además uso de técnicas de vectorización de texto para que este pueda ser tratado por otros modelos de machine learning no orientados al tratamiento de texto. La vectorización del texto tiene un problema y es la pérdida del contexto de las palabras ya que se pierde el orden. Es por esto que esta clase de modelos no es específica ni recomendada para tareas como la nuestra que requiere de clasificar datos que contienen texto.

Modelo RNN - LSTM

El código desarrollado para este modelo se incluye en el Jupyter Notebook: IA-TrabajoFinal-RNN-LSTM-v1.

La estructura del código desarrollado es la siguiente:

1. Cargamos las librerías necesarias para el programa:
 - a. tensorflow.
 - b. tensorflow.keras.models – Sequential
 - c. tensorflow.keras.layers - LSTM, Dense, Embedding
 - d. tensorflow.keras.preprocessing.text – Tokenizer
 - e. tensorflow.keras.preprocessing.sequence – pad_sequence
 - f. sklearn.model_selection – train_test_split
 - g. sklearn.metrics – confusion_matrix, classification_report
2. Cargamos el dataset directamente desde el GitHub del proyecto.
3. Preprocesamos el texto.
 - a. Para ello necesitamos tokenizar el texto. Como podemos ver en este fragmento de texto:

```
# Aseguramos que los valores en las columnas sean cadenas de texto
news["contenido"] = news["title"].astype(str) + " " + news["text"].astype(str) # Concatenamos el título y el texto
tokenizer = Tokenizer(num_words=5000, oov_token=<OOV>) # Creamos el tokenizador
tokenizer.fit_on_texts(news["contenido"]) # Ajustamos el tokenizador a los títulos y texto concatenados
```

Tokenizer(num_words=5000, oov_token=<OOV>). Esta función se utiliza para convertir texto en secuencia de números. Los parámetros que se les pasa son:

- num_words = 5000. Este es el número máximo de palabras que se incluirán en el tokenizador. Las palabras más frecuentes quedarán dentro del límite.
- oov_token = Define el token para palabras fuera del vocabulario. En este caso "<OOV>" se utiliza para representar a palabras desconocidas.
- b. Convertimos el texto a secuencias. El método fit_on_texts() de Tokenizer es utilizado para analizar el texto y crear un vocabulario basado en las palabras encontradas.

```
#Convertimos el texto a secuencias
secuencias = tokenizer.texts_to_sequences(news["contenido"]) # Convertimos el texto a secuencias
secuencias = pad_sequences(secuencias, maxlen=200, padding='post') # Rellenamos las secuencias para que tengan la misma longitud
```

- c. Con pad_sequences() podemos ajustar la longitud de las secuencias numéricas, asegurando que todas tengan la misma longitud. Este ajuste es útil cuando trabajas con redes neuronales, ya que facilita el procesamiento de datos de entrada con una longitud uniforme.

Los parámetros que reciben son:

- secuencias. Es el conjunto de secuencias numéricas que deseas ajustar.
- maxlen=200. Establece la longitud máxima de cada secuencia. Si una secuencia es más corta, se rellena; si es más larga, se recorta.
- padding='post'. Rellena con ceros al final de las secuencias más cortas.
- truncating='post'. Si una secuencia es más larga que maxlen, los valores extra se recortan desde el final.

```
secuencias = pad_sequences(secuencias, maxlen=200, padding='post',
truncating='post')
```

- d. Utilizamos todas las columnas del dataset, ya que tienen información clave para poder evaluar las noticias, por ejemplos, categoría de las noticias (política, social, deportiva), la fuente donde se ha publicado la noticia o supuesta noticia (CNN, BBC, Fox) y la fecha de publicación. Se fusionan todos los datos en la variable “final”.
4. Fase de entrenamiento y prueba del dataset. Aquí hemos realizado dos ejercicios para la división del dataset.
- División simple: 80% de entrenamiento y 20% de prueba. Usamos esta división para evaluar qué configuración del modelo LTSM/GRU es mejor. Una vez
 - División usando métodos KFold, random Cross-Validation, Leave-One-Out Cross-Validation. Nos hemos decantado por el método KFold y a continuación explicamos el porqué:
 - Leave-One-Out Cross-Validation (LOO): Hemos descartado este método porque, a pesar de aprovechar al máximo los datos, es extremadamente costoso en tiempo y el dataset no es pequeño (ya que tenemos un dataset de 18.000 filas).
 - Random Cross-Validation: Tiene una carga computacional más reducida, ya que se realiza una selección aleatoria de muestras. Sin embargo, es menos robusto que, por ejemplo, K-Fold, ya que depende de la aleatoriedad, algunas muestras pueden nunca ser validadas. Lo cual puede sesgar el resultado.
 - K-Fold Cross-Validation:** Permite mejorar la estimación del rendimiento. Usa varias divisiones del dataset, reduciendo la variabilidad en la evaluación. Hace uso de todos los datos del dataset, para entrenamiento y validación. Tiene menos riesgo de sobreajuste, ya que comparado con un solo conjunto de validación, K-Fold distribuye mejor los errores.

Pese a ser más costoso computacionalmente, hemos decidido utilizar el método K-Fold. El valor de K elegido ha sido 5 por los siguientes motivos:

- Balance entre sesgo y varianza. No es demasiado bajo para generar evaluaciones poco confiables ni demasiado alto para sobreajustar el modelo.
- Eficiencia computacional. Se reduce el tiempo de entrenamiento en comparación con valores más altos de K, especialmente en un modelo complejo como LSTM o GRU.
- Uso recomendado en Deep learning. Es una práctica estándar en procesamiento de lenguaje natural (NLP).
- Mejora la generalización. Usa varias participaciones de los datos sin hacer cada conjunto de entrenamiento demasiado pequeño.

5. Evaluamos el modelo.

6. Seguidamente, usamos una nueva noticia creada por nosotros y realizamos una predicción usando el modelo entrenado.
7. Por último, evaluamos el modelo haciendo uso del conjunto de pruebas. Generamos una matriz de confusión y un classification report. Estos dos criterios se utilizarán para evaluar todos los modelos IA utilizados.

Damos más detalle sobre el modelo LSTM y la configuración considerada. La explicación dada a continuación es en base a la división 80% dataset para entrenamiento y 20% para evaluación.

Los parámetros que hemos considerado para la creación del modelo LSTM son:

- Creación del modelo LSTM
 - Número de unidades de LSTM. Hemos evaluado los valores 50 y 100, combinados con Dense. Activación: tanh. Manejo de datos secuenciales de manera estable.
 - Dense. 16, 32 y 64. Activación: relu. Se elige por ser rápida y eficiente en redes profunda
 - Dense. 1. Activación: sigmoid. Produce una salida entre 0 y 1. Se usa en la capa final cuando hay clasificación binaria.
 - Dropout. Ayuda a reducir el riesgo de sobreajuste.
- Entrenamiento del modelo LSTM:
 - epochs (número de veces que el modelo procesa el dataset completo). Se recomienda usar epochs entre 20 y 50.
 - batch_size (número de muestras procesadas antes de actualizar los pesos). Se recomienda usar valores entre 32 y 64.
- Número de unidades de LSTM = 50 y Dense = 16, epochs = 10
 - Entrenamiento:

```

modelo = Sequential([
    Embedding(input_dim=5000, output_dim=64, input_length=secuencias.shape[1]),
    LSTM(50,activation='tanh'),
    Dense(16, activation='relu'), # Procesa categorías y fuentes
    Dense(1, activation='sigmoid') # Capa de salida. Salida binaria (noticia falsa o no)
]) # Modelo secuencial con una capa de embedding, una capa LSTM y dos capas densas

# Compilamos el modelo
modelo.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']) # Compilamos el modelo con el optimizador Adam y la función de pérdida binary_crossentropy

# Entrenamos el modelo
modelo.fit(X_train, y_train, epochs=10, batch_size=32) # Entrenamos el modelo con 10 épocas y tamaño de lote de 32

```

Epoch 1/10
c:/Users/aleja/anaconda3\lib\site-packages\keras\src\layers\core\embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
warnings.warn(
452/452 44s 90ms/step - accuracy: 0.4984 - loss: 0.6936
Epoch 2/10
452/452 43s 94ms/step - accuracy: 0.5156 - loss: 0.6924
Epoch 3/10
452/452 56s 124ms/step - accuracy: 0.5580 - loss: 0.6840
Epoch 4/10
452/452 45s 100ms/step - accuracy: 0.5861 - loss: 0.6701
Epoch 5/10
452/452 42s 93ms/step - accuracy: 0.6170 - loss: 0.6513
Epoch 6/10
452/452 41s 90ms/step - accuracy: 0.6420 - loss: 0.6220
Epoch 7/10
452/452 48s 105ms/step - accuracy: 0.6704 - loss: 0.5916
Epoch 8/10
452/452 239s 530ms/step - accuracy: 0.7041 - loss: 0.5574
Epoch 9/10
452/452 45s 99ms/step - accuracy: 0.7283 - loss: 0.5177
Epoch 10/10
452/452 45s 100ms/step - accuracy: 0.7590 - loss: 0.4804
<keras.src.callbacks.history.History at 0x1a041834fb0>

- Evaluación:

INTELIGENCIA ARTIFICIAL APLICADA A INTERNET DE LAS COSAS. Trabajo Final

```
# Evaluamos el modelo usando solo las secuencias de texto (como en el entrenamiento)
accuracy = modelo.evaluate(secuencias, news["label"])[1]
print(f"Precisión del modelo con todas las columnas y LSTM: {accuracy:.2f}")

564/564 32s 56ms/step - accuracy: 0.4433 - loss: 0.7941
Precisión del modelo con todas las columnas y LSTM: 0.44

nuevo_texto = "El presidente de los Estados Unidos, Joe Biden, anunció hoy nuevas medidas para combatir el cambio climático."
# Convertimos el nuevo texto a secuencias
secuencia = tokenizer.texts_to_sequences([nuevo_texto]) # Convertimos el texto a secuencias
secuencia_padded = pad_sequences(secuencia, maxlen=200, padding='post', truncating='post') # Rellenamos la secuencia para que tenga la misma longitud

## Supongamos que pertenece a la categoría "Política" y es de "CNN"
nuevas_características = np.array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 200, 1500]]) # Simulación de valores

# Concatenamos la secuencia con las nuevas características
entrada_final = np.hstack((secuencia_padded, nuevas_características)) # Fusionamos la secuencia y las nuevas características

# Realizamos la predicción
predicciones = modelo.predict(entrada_final) # Predecimos el nuevo texto
print(f"Predicción para el nuevo texto: {predicciones[0][0]:.2f}") # Mostramos la predicción
# Mostramos la predicción

# convertir la predicción a una etiqueta
Etiqueta = "Noticia falsa" if predicciones[0][0] > 0.5 else "Noticia verdadera"

print(f"Esta es una {Etiqueta}") # Mostramos la etiqueta

1/1 0s 222ms/step
Predicción para el nuevo texto: 0.42
Esta es una Noticia verdadera
```

- Número de unidades de LSTM = 100 y Dense = 16, epochs = 10
 - Entreno

```
modelo = Sequential([
    Embedding(input_dim=5000, output_dim=64, input_length=secuencias.shape[1]),
    LSTM(100, activation='tanh', dropout=0.2), # Capa LSTM con 100 unidades y función de activación tanh. dropout del 20% para evitar el sobreajuste
    Dense(16, activation='relu'), # Procesa categorías y fuentes. probar luego con 64.
    Dense(1, activation='sigmoid') # Capa de salida. Salida binaria (noticia falsa o no)
]) # Modelo secuencial con una capa de embedding, una capa LSTM y dos capas densas

# Compilamos el modelo
modelo.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']) # Compilamos el modelo con el optimizador Adam y la función de pérdida binary_crossentropy

# Entrenamos el modelo
modelo.fit(X_train, y_train, epochs=10, batch_size=32) # Entrenamos el modelo con 10 épocas y tamaño de lote de 32

23m 12.9s

Epoch 1/10
452/452 87s 188ms/step - accuracy: 0.5017 - loss: 0.6936
Epoch 2/10
452/452 78s 172ms/step - accuracy: 0.5025 - loss: 0.6932
Epoch 3/10
452/452 79s 175ms/step - accuracy: 0.5188 - loss: 0.6916
Epoch 4/10
452/452 369s 818ms/step - accuracy: 0.5684 - loss: 0.6775
Epoch 5/10
452/452 77s 139ms/step - accuracy: 0.5886 - loss: 0.6718
Epoch 6/10
452/452 59s 130ms/step - accuracy: 0.5825 - loss: 0.6685
Epoch 7/10
452/452 60s 132ms/step - accuracy: 0.5951 - loss: 0.6627
Epoch 8/10
452/452 58s 128ms/step - accuracy: 0.6189 - loss: 0.6478
Epoch 9/10
452/452 461s 1s/step - accuracy: 0.6444 - loss: 0.6288
Epoch 10/10
452/452 65s 143ms/step - accuracy: 0.6667 - loss: 0.6032

<keras.src.callbacks.history.History at 0x1c0085e56d0>
```

- Evaluación:

INTELIGENCIA ARTIFICIAL APLICADA A INTERNET DE LAS COSAS. Trabajo Final

```
# Evaluamos el modelo usando solo las secuencias de texto (como en el entrenamiento)
accuracy = modelo.evaluate(secuencias, news["label"])[1]
print(f"Precisión del modelo con todas las columnas y LSTM: {accuracy:.2f}")
1 ✓ 31.8s
564/564 32s 56ms/step - accuracy: 0.4433 - loss: 0.7941
Precisión del modelo con todas las columnas y LSTM: 0.44

nuevo_texto = "El presidente de los Estados Unidos, Joe Biden, anunció hoy nuevas medidas para combatir el cambio climático."
# Convertimos el nuevo texto a secuencias
secuencia = tokenizer.texts_to_sequences([nuevo_texto]) # Convertimos el texto a secuencias
secuencia_padded = pad_sequences(secuencia, maxlen=200, padding='post', truncating='post') # Rellenamos la secuencia para que tenga la misma longitud

## Supongamos que pertenece a la categoría "Política" y es de "CNN"
nuevas_características = np.array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 200, 1500]]) # Simulación de valores

# Concatenamos la secuencia con las nuevas características
entrada_final = np.hstack((secuencia_padded, nuevas_características)) # Fusionamos la secuencia y las nuevas características

# Realizamos la predicción
predicciones = modelo.predict(entrada_final) # Predecimos el nuevo texto
print(f"Predicción para el nuevo texto: {predicciones[0][0]:.2f}") # Mostramos la predicción
# Mostramos la predicción

# convertir la predicción a una etiqueta
Etiqueta = "Noticia falsa" if predicciones[0][0] > 0.5 else "Noticia verdadera"

print(f"Esta es una {Etiqueta}") # Mostramos la etiqueta
1 ✓ 0.3s
1/1 0s 222ms/step
Predicción para el nuevo texto: 0.42
Esta es una Noticia verdadera
```

- Número de unidades de LSTM = 100 y Dense = 32, epochs = 10
 - Entreno

```
modelo = Sequential([
    Embedding(input_dim=5000, output_dim=64, input_length=secuencias.shape[1]),
    LSTM(100,activation='tanh', dropout=0.2), # Capa LSTM con 100 unidades y función de activación tanh. dropout del 20% para evitar el sobreajuste
    Dense(32, activation='relu'), # Procesa categorías y fuentes. probar luego con 64.
    Dense(1, activation='sigmoid') # Capa de salida. Salida binaria (noticia falsa o no)
]) # Modelo secuencial con una capa de embedding, una capa LSTM y dos capas densas

# Compilamos el modelo
modelo.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']) # Compilamos el modelo con el optimizador Adam y la función de pérdida binary_crossentropy

# Entrenamos el modelo
modelo.fit(X_train, y_train, epochs=10, batch_size=32) # Entrenamos el modelo con 10 épocas y tamaño de lote de 32
1 ✓ 13m 59s
c:/Users/aleja/anaconda3/lib/site-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
warnings.warn(
Epoch 1/10
452/452 69s 146ms/step - accuracy: 0.4970 - loss: 0.6934
Epoch 2/10
452/452 100s 221ms/step - accuracy: 0.5154 - loss: 0.6930
Epoch 3/10
452/452 75s 165ms/step - accuracy: 0.5516 - loss: 0.6873
Epoch 4/10
452/452 71s 140ms/step - accuracy: 0.5788 - loss: 0.6760
Epoch 5/10
452/452 75s 166ms/step - accuracy: 0.5856 - loss: 0.6697
Epoch 6/10
452/452 84s 187ms/step - accuracy: 0.5935 - loss: 0.6653
Epoch 7/10
452/452 78s 171ms/step - accuracy: 0.6046 - loss: 0.6565
Epoch 8/10
452/452 70s 155ms/step - accuracy: 0.6267 - loss: 0.6406
Epoch 9/10
452/452 88s 195ms/step - accuracy: 0.6572 - loss: 0.6157
Epoch 10/10
452/452 77s 170ms/step - accuracy: 0.6821 - loss: 0.5945
<keras.src.callbacks.history.History at 0x1c00862b200>
```

- Evaluación

INTELIGENCIA ARTIFICIAL APLICADA A INTERNET DE LAS COSAS. Trabajo Final

```
# Evaluamos el modelo usando solo las secuencias de texto (como en el entrenamiento)
accuracy = modelo.evaluate(secuencias, news["label"])[1]
print(f"Precisión del modelo con todas las columnas y LSTM: {accuracy:.2f}")
✓ 31.2s
564/564 31s 55ms/step - accuracy: 0.5816 - loss: 0.6940
Precisión del modelo con todas las columnas y LSTM: 0.58

nuevo_texto = "El presidente de los Estados Unidos, Joe Biden, anunció hoy nuevas medidas para combatir el cambio climático."
# Convertimos el nuevo texto a secuencias
secuencia = tokenizer.texts_to_sequences([nuevo_texto]) # Convertimos el texto a secuencias
secuencia_padded = pad_sequences(secuencia, maxlen=200, padding='post', truncating='post') # Rellenamos la secuencia para que tenga la misma longitud

## Supongamos que pertenece a la categoría "Política" y es de "CNN"
nuevas_características = np.array([[0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 200, 1500]]) # Simulación de valores

# Concatenamos la secuencia con las nuevas características
entrada_final = np.hstack((secuencia_padded, nuevas_características)) # Fusionamos la secuencia y las nuevas características

# Realizamos la predicción
predicciones = modelo.predict(entrada_final) # Predecimos el nuevo texto
print(f"Predicción para el nuevo texto: {predicciones[0][0]:.2f}") # Mostramos la predicción
# Mostramos la predicción

# convertir la predicción a una etiqueta
Etiqueta = "Noticia falsa" if predicciones[0][0] > 0.5 else "Noticia verdadera"

print(f"Esta es una {Etiqueta}") # Mostramos la etiqueta
✓ 0.3s
1/1 0s 265ms/step
Predicción para el nuevo texto: 0.50
Esta es una Noticia verdadera
```

- Número de unidades de LSTM = 100 y Dense = 64, epochs = 10
 - Entreno

```
modelo = Sequential([
    Embedding(input_dim=5000, output_dim=64, input_length=secuencias.shape[1]),
    LSTM(100,activation='tanh', dropout=0.2), # Capa LSTM con 100 unidades y función de activación tanh. dropout del 20% para evitar el sobreajuste
    Dense(64, activation='relu'), # Procesa categorías y fuentes. probar luego con 64,
    Dense(1, activation='sigmoid') # Capa de salida. Salida binaria (noticia falsa o no)
]) # Modelo secuencial con una capa de embedding, una capa LSTM y dos capas densas

# Compilamos el modelo
modelo.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']) # Compilamos el modelo con el optimizador Adam y la función de pérdida binary_crossentropy

# Entrenamos el modelo
modelo.fit(X_train, y_train, epochs=10, batch_size=32) # Entrenamos el modelo con 10 épocas y tamaño de lote de 32
✓ 14m 50.1s

Epoch 1/10
452/452 88s 187ms/step - accuracy: 0.5068 - loss: 0.6935
Epoch 2/10
452/452 109s 241ms/step - accuracy: 0.5119 - loss: 0.6929
Epoch 3/10
452/452 100s 221ms/step - accuracy: 0.5291 - loss: 0.6903
Epoch 4/10
452/452 107s 235ms/step - accuracy: 0.5662 - loss: 0.6787
Epoch 5/10
452/452 112s 246ms/step - accuracy: 0.5923 - loss: 0.6676
Epoch 6/10
452/452 72s 158ms/step - accuracy: 0.6021 - loss: 0.6627
Epoch 7/10
452/452 70s 155ms/step - accuracy: 0.5889 - loss: 0.6727
Epoch 8/10
452/452 71s 158ms/step - accuracy: 0.6209 - loss: 0.6480
Epoch 9/10
452/452 73s 161ms/step - accuracy: 0.6519 - loss: 0.6234
Epoch 10/10
452/452 90s 198ms/step - accuracy: 0.6864 - loss: 0.5964

<keras.src.callbacks.history.History at 0x1c019bcaba0>
```

- Evaluación

```

# Evaluamos el modelo usando solo las secuencias de texto (como en el entrenamiento)
accuracy = modelo.evaluate(secuencias, news["label"])[1]
print(f"Precisión del modelo con todas las columnas y LSTM: {accuracy:.2f}")
✓ 37.8s
564/564 ━━━━━━━━ 38s 66ms/step - accuracy: 0.4283 - loss: 0.8308
Precisión del modelo con todas las columnas y LSTM: 0.43

nuevo_texto = "El presidente de los Estados Unidos, Joe Biden, anunció hoy nuevas medidas para combatir el cambio climático."
# Convertimos el nuevo texto a secuencias
secuencia = tokenizer.texts_to_sequences([nuevo_texto]) # Convertimos el texto a secuencias
secuencia_padded = pad_sequences(secuencia, maxlen=200, padding='post', truncating='post') # Rellenamos la secuencia para que tenga la misma longitud
## Supongamos que pertenece a la categoría "Política" y es de "CNN"
nuevas_características = np.array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 200, 1500]]) # Simulación de valores

# Concatenamos la secuencia con las nuevas características
entrada_final = np.hstack((secuencia_padded, nuevas_características)) # Fusionamos la secuencia y las nuevas características

# Realizamos la predicción
predicciones = modelo.predict(entrada_final) # Predecimos el nuevo texto
print(f"Predicción para el nuevo texto: {predicciones[0][0]:.2f}") # Mostramos la predicción
# Mostramos la predicción

# convertir la predicción a una etiqueta
Etiqueta = "Noticia falsa" if predicciones[0][0] > 0.5 else "Noticia verdadera"

print(f"Esta es una {Etiqueta}") # Mostramos la etiqueta
✓ 0.2s
1/1 ━━━━━━ 0s 205ms/step
Predicción para el nuevo texto: 0.55
Esta es una Noticia falsa

```

En base a esta información, no vemos gran diferencia de accuracy y loss entre incrementar el número de unidades LSTM y Dense. Por ello, hemos optado por elegir la siguiente configuración para la creación del modelo LSTM.

```

modelo = Sequential([
    Embedding(input_dim=5000, output_dim=64, input_length=secuencias.shape[1]),
    LSTM(50, activation='tanh', dropout=0.2), # Capa LSTM con 50 unidades y función de activación tanh. dropout del 20% para evitar el sobreajuste
    Dense(16, activation='relu'), # Procesa categorías y fuentes. relu es la función de activación que se utiliza para la capa oculta
    Dense(1, activation='sigmoid') # Capa de salida. Salida binaria (noticia falsa o no)
]) # Modelo secuencial con una capa de embedding, una capa LSTM y dos capas densas

```

Con respecto a la elección de epochs y batch_size, hemos podido ver que:

- epochs. Los valores asociados a accuracy y loss entre epochs = 10 y epochs = 35 aumentan y disminuyen, respectivamente. Sin embargo, a la hora de evaluar el modelo con el conjunto de prueba, la accuracy y loss no aumenta significativamente.

A continuación, mostramos los valores de accuracy y loss obtenidos para epochs = 35 y batch_size = 32.

```

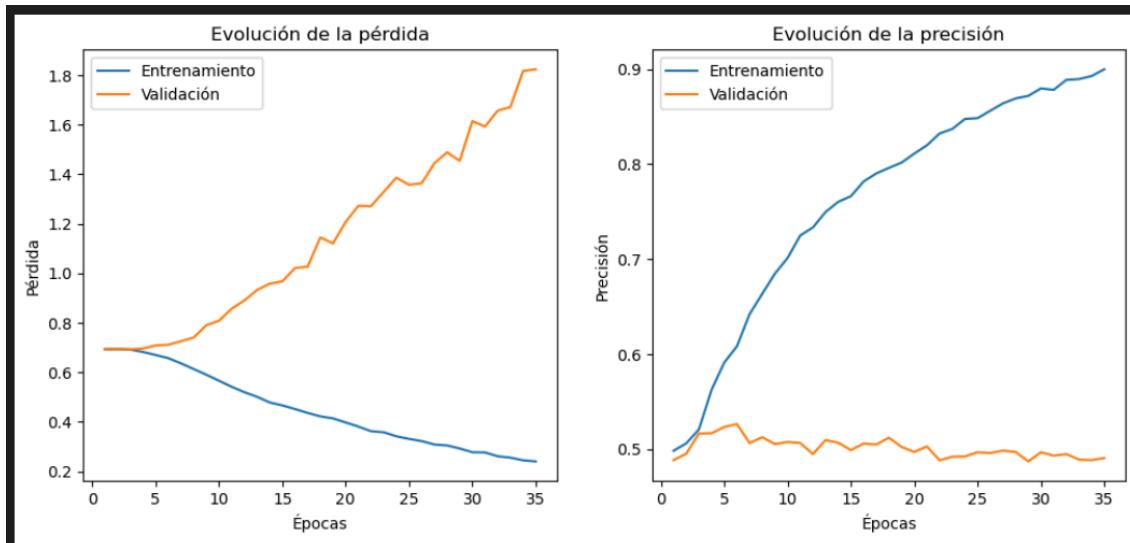
Epoch 32/35
452/452 ━━━━━━━━ 118s 185ms/step - accuracy: 0.8930 - loss: 0.2498 - val_accuracy: 0.4949 - val_loss: 1.6574
Epoch 33/35
452/452 ━━━━━━ 99s 218ms/step - accuracy: 0.8978 - loss: 0.2383 - val_accuracy: 0.4891 - val_loss: 1.6717
Epoch 34/35
452/452 ━━━━━━ 112s 153ms/step - accuracy: 0.8973 - loss: 0.2368 - val_accuracy: 0.4885 - val_loss: 1.8174
Epoch 35/35
452/452 ━━━━━━ 79s 174ms/step - accuracy: 0.9053 - loss: 0.2295 - val_accuracy: 0.4907 - val_loss: 1.8245

```

Como vemos en el siguiente gráfico de pérdida, la línea azul (entrenamiento) disminuye constantemente, lo que indica que el modelo está aprendiendo bien en los datos de entrenamiento. La línea naranja (validación) aumenta, lo que sugiere que el modelo no está generalizando bien a datos nuevos.

En base a esto, podemos considerar que el modelo está memorizando los datos en lugar de aprender patrones útiles. Esto es un indicio de sobreajuste.

Con respecto al gráfico de la precisión, la línea azul (entrenamiento) sube, lo que indica que el modelo se vuelve más preciso en los datos de entrenamiento. Sin embargo, la línea naranja (validación) se mantiene constante, lo que muestra que la precisión en datos nuevos no mejora. Esto puede ser un indicio de que el modelo no está aprendiendo bien características relevantes.



Para evitar un sobreajuste y mejora de rendimiento en validación, hemos optado por aplicar lo siguiente:

- Aumentar dropout a 0.3. Esto significa que se “apaga” un 30% de las unidades durante el entrenamiento, con el objetivo de evitar que el modelo dependa demasiado de ciertos patrones existentes en los datos de entrenamiento. Así se “obliga” al modelo a que aprenda características más generales, útil para mejorar el rendimiento en datos nuevos, y evita que el modelo memorice el dataset. 0,2 es un estándar recomendado en redes recurrentes, ya que un valor bajo (<0.1) puede que no evite el sobreajuste y un valor alto (>0.5) podría eliminar demasiada información y afectar el aprendizaje.
- Usar Early Stopping para detener el entrenamiento si la validación deja de mejorar.
- batch_size. Hemos probado con 32. Al estar dentro del rango de recomendación indicado.

En resumen y en base a lo explicado anteriormente, hemos aplicado esta configuración:

```

modelo = Sequential([
    Embedding(input_dim=5000, output_dim=64, input_length=secuencias.shape[1]),
    LSTM(50, activation='tanh', dropout=0.2), # Capa LSTM con 50 unidades y función de activación tanh. dropout del 20% para evitar el sobreajuste
    Dense(16, activation='relu'), # Procesa categorías y fuentes. relu es la función de activación que se utiliza para la capa oculta
    Dense(1, activation='sigmoid') # Capa de salida. Salida binaria (noticia falsa o no)
]) # Modelo secuencial con una capa de embedding, una capa LSTM y dos capas densas

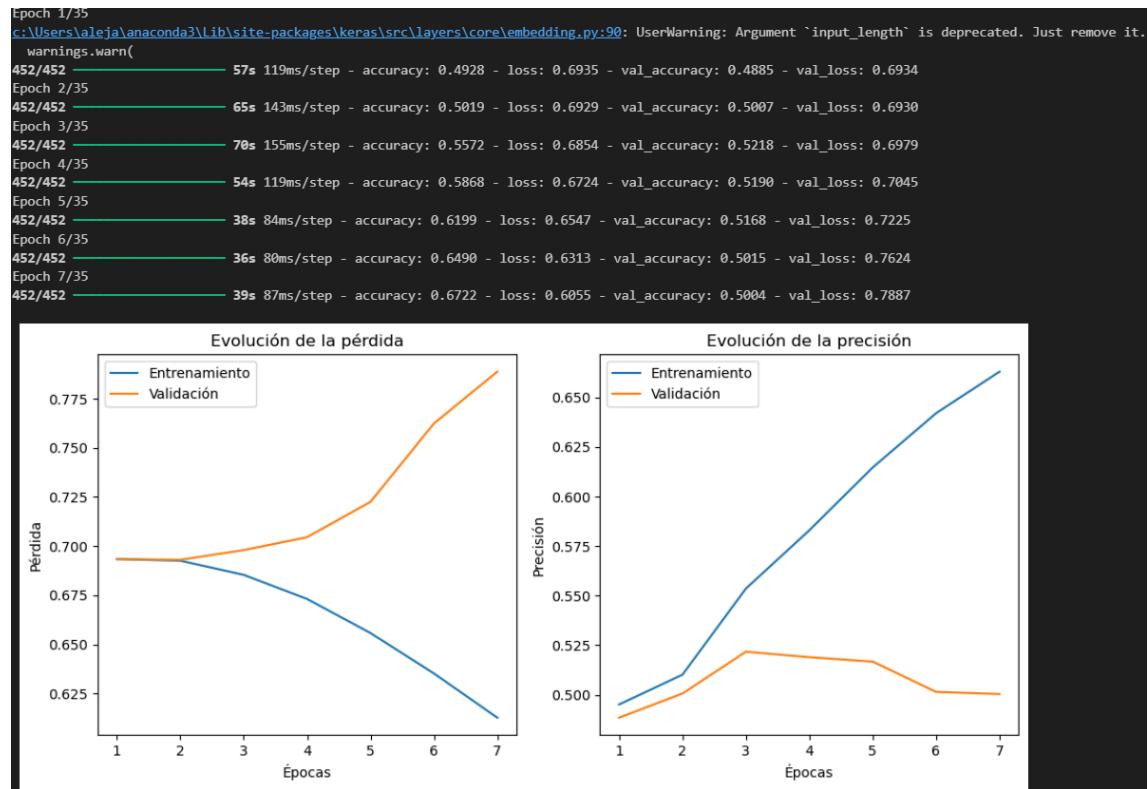
# Compilamos el modelo
modelo.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']) # Compilamos el modelo con el optimizador Adam y la función de pérdida binary_crossentropy

early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True) # Early stopping para evitar el sobreajuste. Monitorea la pérdida de validación y restaura los mejores pesos

# Entrenamos el modelo con datos de validación
historial = modelo.fit(X_train, y_train, epochs=35, batch_size=32, validation_data=(X_test, y_test), callbacks=[early_stop]) # Entrenamos el modelo con 35 épocas y tamaño de lote de 32.

```

Tras aplicar el Early Stopping, podemos ver que se para el entrenamiento en el epoch = 7.



Vemos que el accuracy en la evaluación es del 0.5020.

```

# Evaluamos el modelo usando solo las secuencias de texto (como en el entrenamiento)
accuracy = modelo.evaluate(secuencias, news["label"])[1] # Evaluamos el modelo con las secuencias de texto. news["label"] es la variable objetivo.
print(f"Precisión del modelo con todas las columnas y LSTM: {accuracy:.2f}")
✓ 17.7s
564/564      18s 30ms/step - accuracy: 0.5020 - loss: 0.6935
Precisión del modelo con todas las columnas y LSTM: 0.51

```

Por último, probamos con una supuesta noticia falsa. Definimos el umbral para etiquetar los resultados. Tras probar 0.4, 0.5 y 0.6, hemos considerado quedarnos con 0.4, ya que esto nos ayuda a reducir los falsos negativos, priorizar la identificación de la mayoría de las noticias falsas, aunque aumente el riesgo de algunos falsos positivos.

Los valores obtenidos para esta noticia: "Joe Biden, Presidente de EEUU, ha dimitido." son los siguientes:

```

1/1 ----- 0s 69ms/step
Predicción para el nuevo texto: 0.50
Esta es una Noticia falsa
113/113 ----- 3s 29ms/step
Matriz de confusión:
[[ 0 1846]
 [ 0 1763]]
Explicación de la matriz de confusión:
[TN, FP]
[FN, TP]
Reporte de clasificación:
      precision    recall   f1-score   support
          0         0.00     0.00     0.00      1846
          1         0.49     1.00     0.66      1763

accuracy                           0.49      3609
macro avg       0.24     0.50     0.33      3609
weighted avg    0.24     0.49     0.32      3609

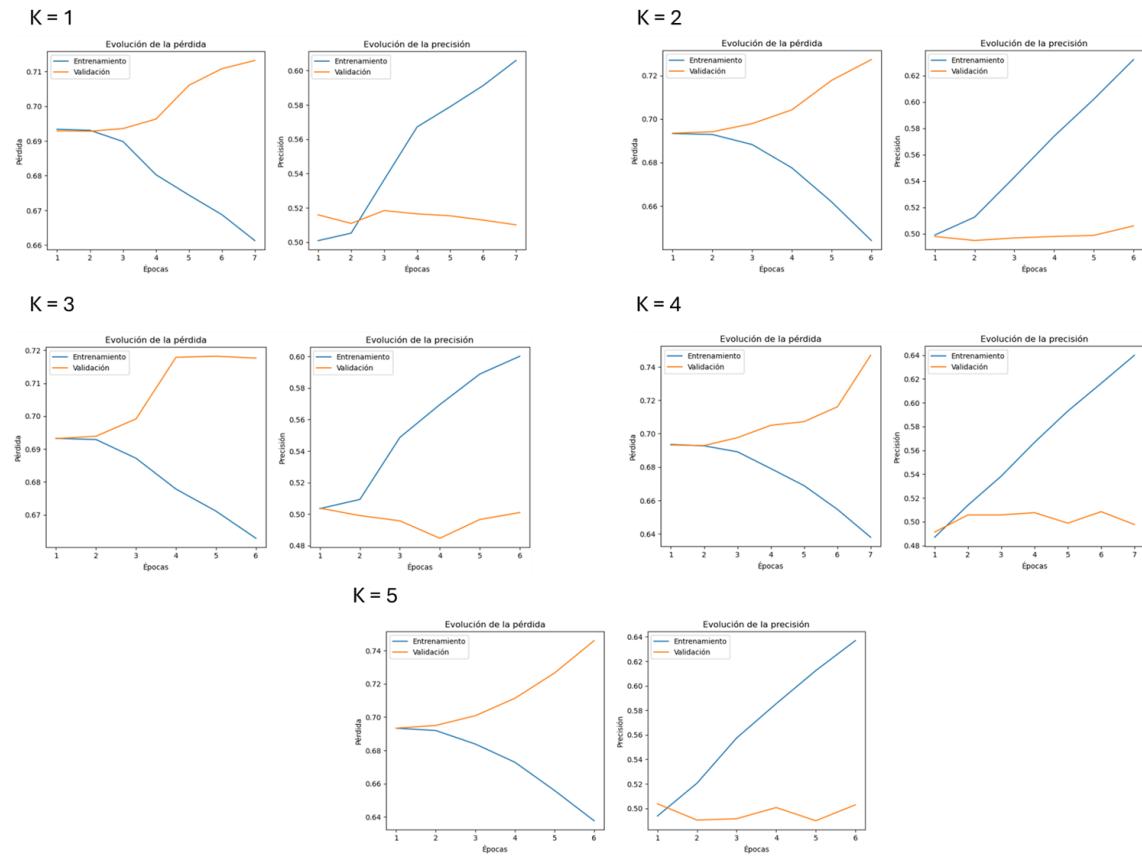
```

RNN-LSTM, K-Fold Cross-Validation, K = 5

A continuación, ofrecemos la información asociada a la ejecución del modelo LSTM para KFold con K = 5.

Durante la programación se nos planteó la pregunta de si deberíamos definir el modelo dentro o fuera del bucle. Se podría definir fuera del bucle y reentrenarlo en cada iteración de éste, sin embargo, hemos podido ver que la razón principal para incluirlo dentro del bucle es la de evitar el sesgo por entrenamiento previo. Esto se produciría porque los pesos anteriores influyen en cada nueva evaluación, afectando a la imparcialidad del resultado. La solución a esto sería el reseteo de los pesos, pero esto se puede simplificar incluyendo el modelo dentro de dentro del bucle.

Como podemos ver en las siguientes gráficas de evolución de pérdida y precisión, entre las épocas 2 y 3 el modelo comienza a demostrar sobreajuste. La precisión media entre ambas épocas ronda el 0.51 y la pérdida el 0.69.



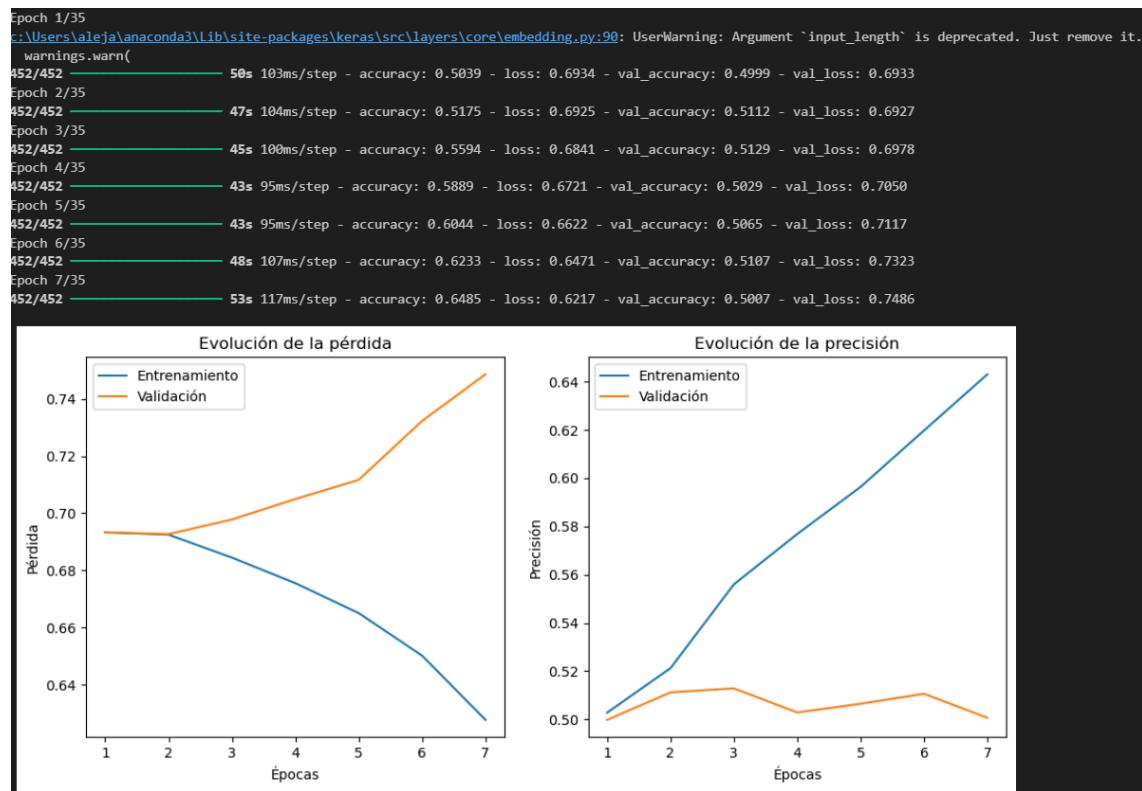
Matriz de confusión y classification report.

```
[0.5006927251815796, 0.4973676800727844, 0.5034635663032532, 0.5023552179336548, 0.4876697063446045]
Precisión promedio con K-Fold Cross Validation (K=5): 0.50
1/1 ━━━━━━ 1s 519ms/step
Predicción para el nuevo texto: 0.50
Esta es una Noticia falsa
113/113 ━━━━━━ 4s 32ms/step
Matriz de confusión:
[[ 0 1809]
 [ 0 1800]]
Explicación de la matriz de confusión:
[TN, FP]
[FN, TP]
Reporte de clasificación:
      precision    recall   f1-score   support
          0         0.00     0.00     0.00     1809
          1         0.50     1.00     0.67     1800

  accuracy                           0.50     3609
   macro avg       0.25     0.50     0.33     3609
weighted avg       0.25     0.50     0.33     3609
```

Modelo RNN - GRU

Aplicando la misma configuración que para LSTM, vemos que se obtiene una gráfica muy similar para la división de 80% de la muestra para entrenamiento y 20% para evaluación.



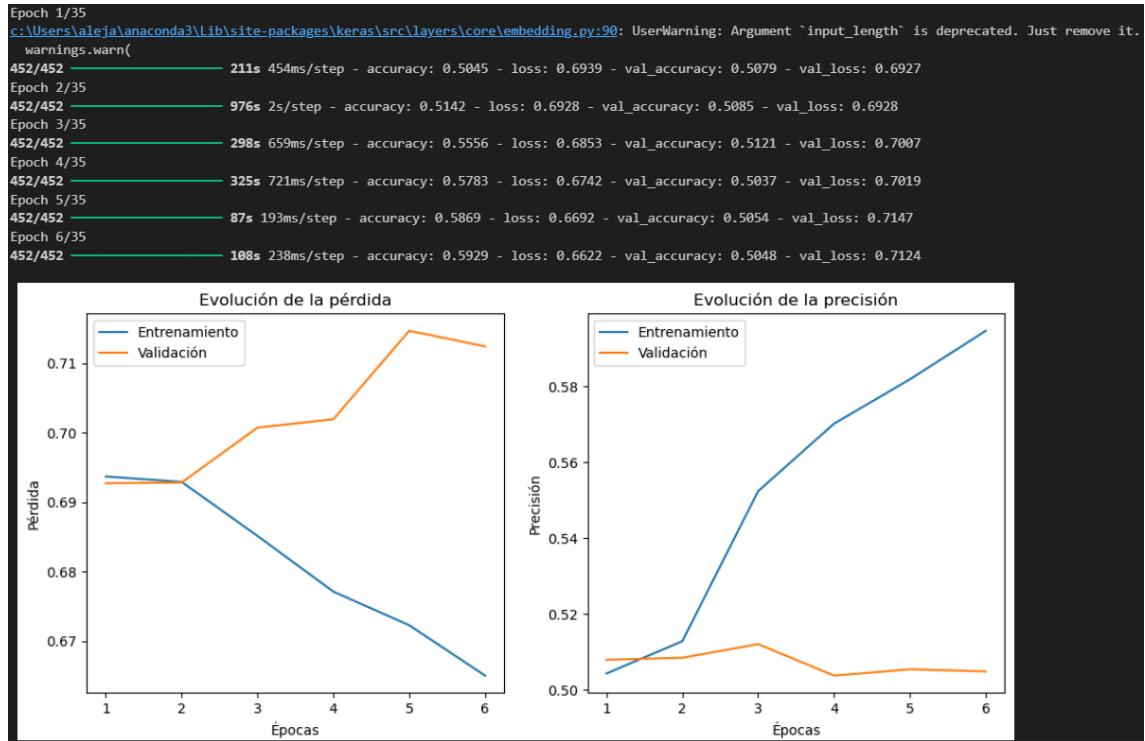
Por ello, modificamos la configuración con el objetivo de poder ver si hay una diferencia y mejora en el proceso de clasificación.

Cambiamos los dos siguientes parámetros:

- Números de unidades GRU = 100.
- Dense = 32.

Podemos ver que, a efectos prácticos, el resultado obtenido con esta nueva configuración es prácticamente la misma, aunque hay una entrecruzado entre entrenamiento y validación, en los resultados tanto para evolución de la precisión como la pérdida.

INTELIGENCIA ARTIFICIAL APLICADA A INTERNET DE LAS COSAS. Trabajo Final



La matriz de confusión y el informe de clasificación para esta configuración es la que sigue:

```

1/1      0s 499ms/step
Predicción para el nuevo texto: 0.49
Esta es una Noticia falsa
113/113    9s 78ms/step
Matriz de confusión:
[[ 0 1846]
 [ 0 1763]]
Explicación de la matriz de confusión:
[TN, FP]
[FN, TP]
Reporte de clasificación:
      precision    recall   f1-score   support
          0         0.00     0.00     0.00      1846
          1         0.49     1.00     0.66      1763

accuracy                           0.49      3609
macro avg       0.24     0.50     0.33      3609
weighted avg    0.24     0.49     0.32      3609

```

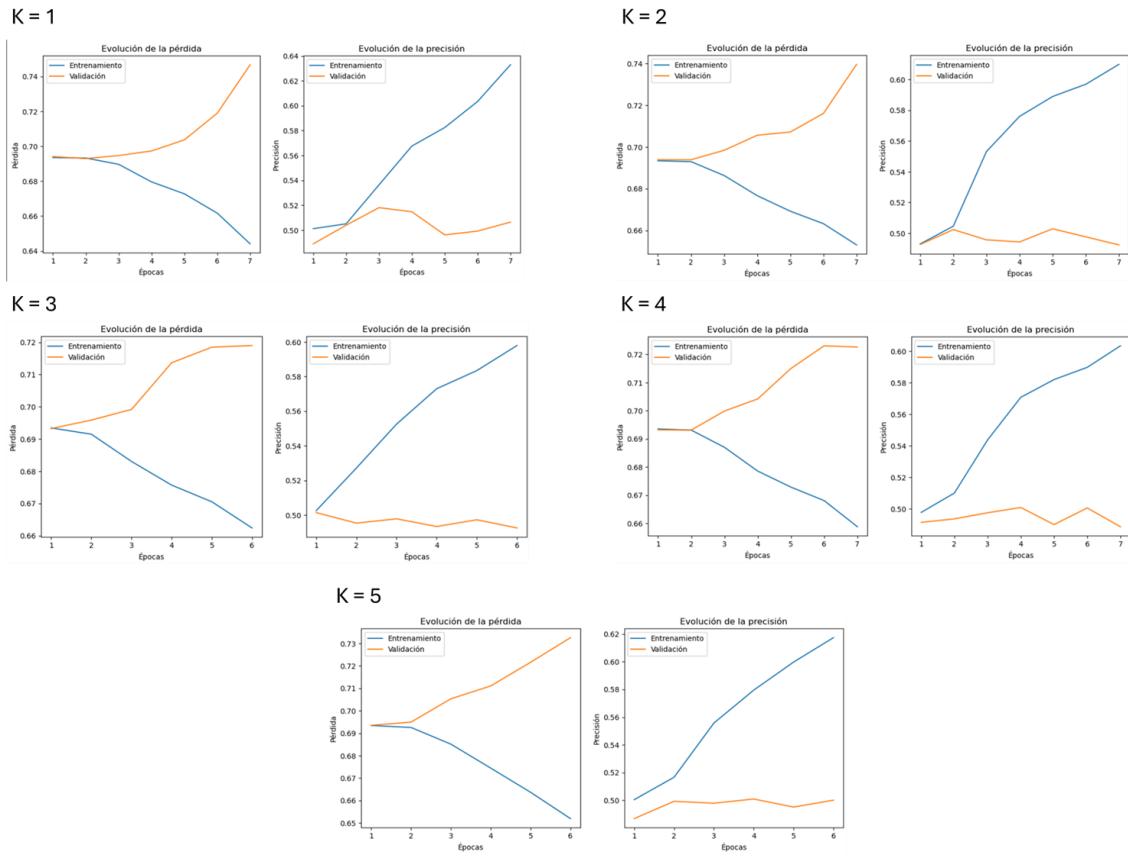
RNN-GRU, K-Fold Cross-Validation, K = 5

A continuación, ofrecemos la información asociada a la ejecución del modelo GRU para KFold con K = 5.

Como explicamos para LSTM, hemos definido el modelo dentro del bucle for, aunque suponga un incremento del coste computacional.

Como podemos ver en las siguientes gráficas de evolución de pérdida y precisión, entre las épocas 2 y 3 el modelo comienza a demostrar sobreajuste. La precisión media entre ambas épocas ronda el 0.51 y la pérdida el 0.69.

También, podemos ver una similitud significativa con las gráficas obtenidas para K-Fold Cross-Validation de LSTM.



```
Precisión promedio con K-Fold Cross Validation (K=5): nan
1/1 ━━━━━━━━ 0s 311ms/step
Predicción para el nuevo texto: 0.50
Esta es una Noticia falsa
113/113 ━━━━━━━━ 5s 42ms/step
Matriz de confusión:
[[ 0 1809]
 [ 0 1800]]
Explicación de la matriz de confusión:
[TN, FP]
[FN, TP]
Reporte de clasificación:
      precision    recall   f1-score   support
          0         0.00     0.00     0.00      1809
          1         0.50     1.00     0.67      1800

accuracy                           0.50      3609
macro avg       0.25     0.50     0.33      3609
weighted avg    0.25     0.50     0.33      3609
```

Modelo BERT

El modelo de transformers para la clasificación de noticias usado ha sido uno basado en una familia de transformers llamada BERT. En concreto se ha usado BertTokenaizer para el encoding del texto en tokens y BertForSequenceClassification para el modelo de clasificación.

Cabe resaltar que la ejecución de este tipo de modelos es muy costosa y se requiere de una GPU para poder ejecutarlos de manera eficiente. Esto ha causado problemas durante el desarrollo de este modelo de clasificación debido a los largos tiempos de entrenamiento, lo cual ha llevado a no poder hacer un gran número de pruebas con varios parámetros.

El primero de los pasos que se ha dado para lograr un modelo de BERT ha sido usar solo los campos de texto de los que disponemos en el dataset para crear nuestro modelo de clasificación. Esto ha sido con el fin de ver cómo se comporta este solo contando con esos parámetros y hacernos una idea si contar con el resto de datos del dataset puede ayudar a mejorar la clasificación.

Lo primero que haremos para poder codificar el texto, será unir los campos de **title** y **text** del dataset.

```
df = pd.read_csv("fake_news_dataset.csv")

df_reducido = df[["title", "text", "label"]].copy()
df_reducido["combined"] = df_reducido["title"] + "[SEP]" + df_reducido["text"]
df_reducido['label'] = df_reducido['label'].map({'fake': 0, 'real': 1})
df_reducido.drop(columns=["title", "text"], inplace=True)

display(df_reducido.head())
```

Para ello hacemos uso del dataset original ya que BERT es capaz de tratar el mismo las stopwords del texto, tratar en minúscula si así lo queremos, por tanto no hace falta hacer uso del texto formateado que hemos generado.

Una vez tenemos preparado el dataframe con el texto y los valores para la clasificación, hacemos uso de un tokenizer para poder transformar el texto en tokens que luego podamos aplicar a nuestro transformer. Para esto, además vamos a dividir el conjunto de datos en un 80% para entrenamiento y un 20% para clasificación con un train_split. Esto lo hacemos mediante este método, ya que contamos con gran cantidad de datos y además la cantidad de noticias falsas y verdaderas es muy pareja por lo tanto los datos de entrenamiento como los de test están muy equilibrados y es un método apenas costoso.

```
# Dividir en entrenamiento y validación
X_train, X_test, y_train, y_test = train_test_split(
    df_reducido["combined"].tolist(), df_reducido["label"].tolist(), test_size=0.2
)

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased", do_lower_case=True)
train_encodings = tokenizer(X_train, truncation=True, padding=True, max_length=500)
test_encodings = tokenizer(X_test, truncation=True, padding=True, max_length=5000)

class NewsDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item["labels"] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = NewsDataset(train_encodings, y_train)
test_dataset = NewsDataset(test_encodings, y_test)
```

Una vez se obtiene el tokenizer y se divide el dataframe, se codifican ambos dataframe, quedando uno para entrenamiento y otro para test ambos listo para poder ser usados por el modelo de BERT.

```
for epoch in range(3):
```

Para entrenar el modelo, y debido a lo costoso de entrenar este tipo de modelos, optamos por lanzar 3 epoch. Para ello cargamos nuestro modelo de BERT antes mencionado y usamos AdamW como finetunner de este, además aplicaremos un tamaño de batch de 16 ya que si aumentamos demasiado el tamaño de batch, pese a que sería lo más óptimo por lo visto en otro modelos, esto requiere de una potencia de procesamiento de la que no disponemos.

```
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
model.to(device)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
optim = AdamW(model.parameters(), lr=1e-8)
```

Una vez tenemos nuestro modelo y finetunner listo, lanzamos el entrenamiento. En este nos damos cuenta de que los distintos batch dentro de las etapas, tienden a ir perdiendo precisión y aumentar la pérdida del modelo a las pocas ejecuciones.

Batch	Avg Loss	Running Accuracy
10	0.7153	0.4750
20	0.7028	0.5188
30	0.6960	0.5354
40	0.6985	0.5312
50	0.7000	0.5275
60	0.6978	0.5344
70	0.6989	0.5312
80	0.6982	0.5281
90	0.6995	0.5188
100	0.6994	0.5200
110	0.6993	0.5193
120	0.6993	0.5172
130	0.6982	0.5188
140	0.6985	0.5210
150	0.6992	0.5200
160	0.6988	0.5195
170	0.6993	0.5147
180	0.7000	0.5111
190	0.7001	0.5099
200	0.7000	0.5106
210	0.6994	0.5134
220	0.6995	0.5131
230	0.6998	0.5125
.		
390	0.7000	0.4830
400	0.7000	0.4833

Es por esto, que se decide hacer uso de la técnica de early-stoppage, configurando una paciencia de 10 ejecuciones para determinar que el modelo está perdiendo precisión y que por tanto hay que parar el entrenamiento.

```
# Configuración de Early Stopping
early_stop_patience = 10 # Número de épocas/batches sin mejora antes de parar
min_delta = 0.001 # Cambio mínimo para considerar mejora
best_loss = float('inf')
no_improve_count = 0
stop_training = False
```

Con esto volvemos a lanzar el modelo, esta vez con early-stoppage, lo cual nos ayudará además a reducir los tiempos de ejecución de hasta 45 minutos que podía llegar a durar el entrenamiento. Tras esto los resultados del entrenamiento fueron los siguientes.

```
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and ar
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Epoch 1/3
Batch 10 | Running Avg Loss: 0.6909 | Running Accuracy: 0.5437
Batch 20 | Running Avg Loss: 0.6923 | Running Accuracy: 0.5312
Batch 30 | Running Avg Loss: 0.6933 | Running Accuracy: 0.5250
Batch 40 | Running Avg Loss: 0.6960 | Running Accuracy: 0.5094
Batch 50 | Running Avg Loss: 0.6959 | Running Accuracy: 0.5088
Batch 60 | Running Avg Loss: 0.6937 | Running Accuracy: 0.5125
Batch 70 | Running Avg Loss: 0.6949 | Running Accuracy: 0.5089
Batch 80 | Running Avg Loss: 0.6953 | Running Accuracy: 0.5117
Batch 90 | Running Avg Loss: 0.6948 | Running Accuracy: 0.5125
Batch 100 | Running Avg Loss: 0.6953 | Running Accuracy: 0.5094
Batch 110 | Running Avg Loss: 0.6953 | Running Accuracy: 0.5074

Early stopping at batch 110 - No improvement for 10 checks

Epoch 1 Summary | Loss: 0.0765 | Accuracy: 0.5074

Early stopping triggered!
```

Como se pudo comprobar en varias ocasiones, con solo el texto el modelo de BERT aplicado nunca lograba superar los 54% de precisión y siempre rondaba por valores cercanos. Además este estancamiento se produce de forma rápida y siempre en la primera etapa.

Tras haber entrenado el modelo, se prueba los resultados de este realizando una predicción con los datos de prueba previamente preparados.

```
test_loader = DataLoader(test_dataset, batch_size=16)

all_preds = []
all_labels = []

with torch.no_grad():
    for batch in test_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        outputs = model(input_ids, attention_mask=attention_mask)
        logits = outputs.logits
        preds = torch.argmax(logits, dim=1)

        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

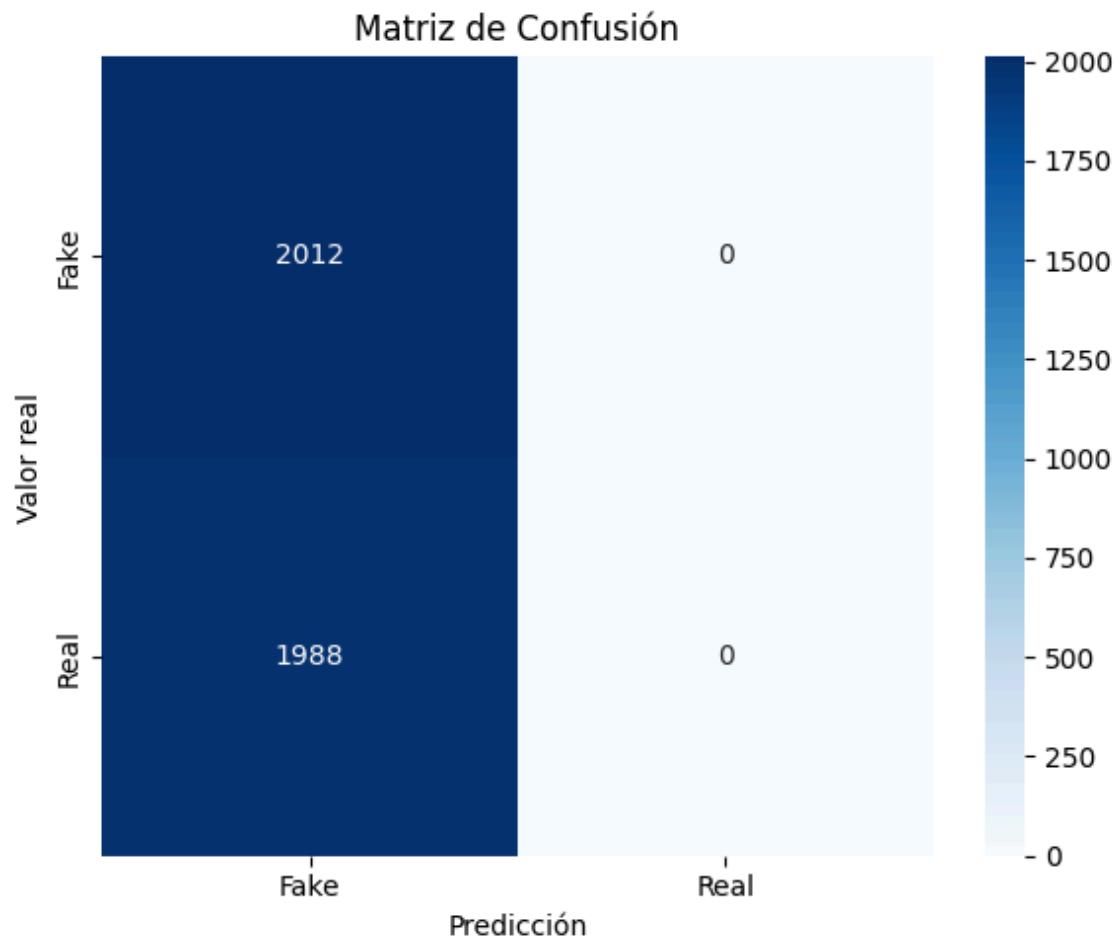
# Matriz de confusión
cm = confusion_matrix(all_labels, all_preds)
print(classification_report(all_labels, all_preds))

# Etiquetas opcionales si tus clases no son solo 0 y 1
labels = ["Fake", "Real"] # reemplázalo si usas otras clases

# Dibujar
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=labels, yticklabels=labels)
plt.xlabel("Predicción")
plt.ylabel("Valor real")
plt.title("Matriz de Confusión")
plt.tight_layout()
plt.show()
```

Estos debido a los datos arrojados por el entrenamiento, no llamaban a dar muy buenos resultados, siendo estos los siguientes.

	precision	recall	f1-score	support
0	0.50	1.00	0.67	2012
1	0.00	0.00	0.00	1988
accuracy			0.50	4000
macro avg	0.25	0.50	0.33	4000
weighted avg	0.25	0.50	0.34	4000



Como se puede observar, todas las predicciones, dieron que las noticias eran falsas, esto hace que el modelo tenga una precisión de 0.50, sin embargo esto se debe más al número de noticias falsas realmente en el dataframe de test que en la clasificación del modelo, ya que este tiende a dar las noticias como falsas. Pasa lo mismo para el caso del f1 score que debido a que el número de noticias falsas del dataframe de test era alto, este tiene un valor de 0.67 no reflejando el mal resultado de la clasificación del modelo sesgado al alto valor del recall al dar toda noticia por falsa. Sin embargo para las noticias verdaderas si que vemos un resultado que refleja mejor la realidad siendo todo 0 ya que no es capaz de clasificar ninguna noticia como verdadera.

Tras esta prueba y con estos resultados decepcionantes, lo siguiente era probar a realizar otro modelo de BERT, esta vez usando todos los datos disponibles para ver si esto podía mejorar las clasificaciones obtenidas.

```

df_encoded = pd.read_csv("datos_limpios_estandarizados.csv")
df_encoded[['title', 'text']] = df_encoded[['title', 'text']].fillna("")

text_data = (df_encoded['title'] + " " + df_encoded['text']).tolist()
numeric_features = df_encoded.drop(columns=['title', 'text', 'label']).astype('float32').values
labels = df_encoded['label'].values

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')

encodings = tokenizer(text_data, truncation=True, padding=True, max_length=512, return_tensors='pt')

X_text_train, X_text_test, X_num_train, X_num_test, y_train, y_test = train_test_split(
    text_data, numeric_features, labels,
    test_size=0.2,           # 20% para prueba
    random_state=42,         # Para reproducibilidad
    stratify=labels          # Mantiene proporción de clases
)

encodings_train = tokenizer(X_text_train, truncation=True, padding=True, max_length=5000, return_tensors='pt')
encodings_test = tokenizer(X_text_test, truncation=True, padding=True, max_length=5000, return_tensors='pt')

```

Para preparar el transformer haciendo uso de todos los datos, cargamos el dataframe que se preparó previamente, haciendo uso del que posee los datos estandarizados. Para el encoding, volvemos a hacer uso del ber-base-uncased, para que el texto sea tratado todo en minúsculas. El resto de pasos son iguales que en la anterior prueba, se hace uso de test_split y se codifican ambos dataframes. Además, separamos previo a esto los campos de texto de los campos numéricos para poder realizar el encoding y luego poder volverlos a juntar en un nuevo dataframe.

```

class NewsDataset(Dataset):
    def __init__(self, encodings, numeric_feats, labels):
        self.encodings = encodings
        self.numeric_feats = torch.tensor(numeric_feats, dtype=torch.float32)
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __getitem__(self, idx):
        item = {key: val[idx] for key, val in self.encodings.items()}
        item['numeric_feats'] = self.numeric_feats[idx]
        item['labels'] = self.labels[idx]
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = NewsDataset(encodings_train, X_num_train, y_train)
test_dataset = NewsDataset(encodings_test, X_num_test, y_test)

```

Generamos los dos dataframes que serán necesarios para el entrenamiento y posterior predicción, juntando los datos numéricos con los texto tokenizados.

Tras esto y para poder ser capaces de usar las características de texto junto con las numéricas, debemos de definir un modelo que haga uso de un transformer de BERT para la clasificación del texto, al que luego se le unirá un modelo de red de neuronas con capa de activación relu para poder juntar la clasificación del texto a las características numéricas del dataframe.

```

class BertWithNumericFeatures(nn.Module):
    def __init__(self, text_model_name, num_numeric_feats):
        super().__init__()
        self.bert = BertModel.from_pretrained(text_model_name)
        self.dropout = nn.Dropout(0.3)
        self.classifier = nn.Sequential(
            nn.Linear(self.bert.config.hidden_size + num_numeric_feats, 128),
            nn.ReLU(),
            nn.Linear(128, 2) # Para clasificación binaria
        )

    def forward(self, input_ids, attention_mask, numeric_feats, labels=None):
        bert_outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        cls_output = bert_outputs.pooler_output # [batch, hidden_size]

        combined = torch.cat((cls_output, numeric_feats), dim=1)
        logits = self.classifier(self.dropout(combined))

        if labels is not None:
            loss_fn = nn.CrossEntropyLoss()
            loss = loss_fn(logits, labels)
            return {"loss": loss, "logits": logits}
        else:
            return {"logits": logits}

```

Una vez tenemos nuestro modelo híbrido preparado, entrenamos el modelo y vemos los resultados del entrenamiento. De nuevo para este usaremos 3 epochs y un batch size de 16.

```

training_args = TrainingArguments(
    output_dir='./results',
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    do_eval=True,
    logging_dir='./logs',
)

def compute_metrics(p):
    from sklearn.metrics import accuracy_score, precision_recall_fscore_support
    preds = p.predictions.argmax(-1)
    labels = p.label_ids
    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds, average='binary')
    acc = accuracy_score(labels, preds)
    return {"accuracy": acc, "f1": f1, "precision": precision, "recall": recall}

trainer = Trainer(
    model=BertWithNumericFeatures('bert-base-uncased', numeric_features.shape[1]),
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset, # deberías separar train/test idealmente
    compute_metrics=compute_metrics
)

trainer.train()

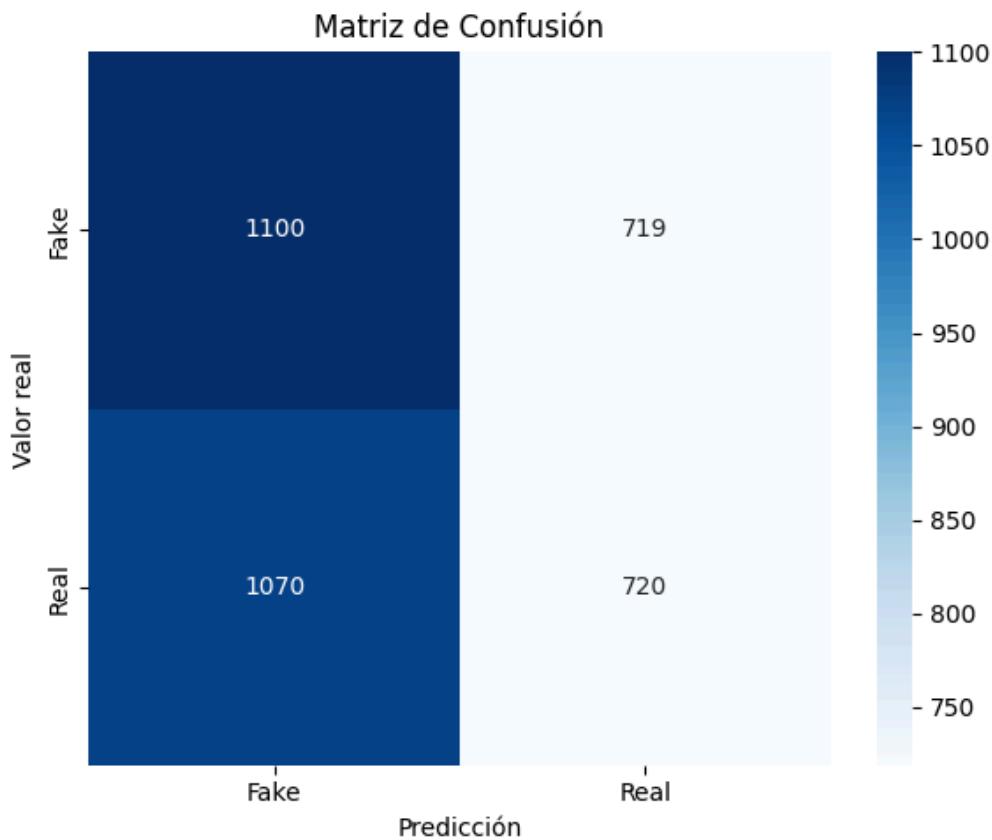
```

Step	Training Loss
500	0.696000
1000	0.694300
1500	0.693400
2000	0.693300
2500	0.694000

Como podemos ver, al igual que pasaba con el modelo anterior, la pérdida se mantiene muy estable y por tanto también lo hace la precisión del modelo. Lo cual nos muestra que se podría hacer al igual para este un early-stoppage, ya que tiende a estancarse el entrenamiento de este.

Una vez entrenado el modelo, se comprueban las predicciones de este.

Reporte de clasificación:					
	precision	recall	f1-score	support	
0	0.51	0.60	0.55	1819	
1	0.50	0.40	0.45	1790	
accuracy			0.50	3609	
macro avg	0.50	0.50	0.50	3609	
weighted avg	0.50	0.50	0.50	3609	



Podemos ver, como al contar con más datos, el modelo ha mejorado, solucionando el problema de la clasificación de noticias como todas falsas, pero sigue habiendo una tendencia a clasificar las noticias como falsas. Los resultados siguen siendo bastante malos, teniendo precisiones de entorno al 50% en ambas clases, sin embargo supone una mejora considerable respecto al modelo anterior.

Modelos no PLN

Además de los modelos PLN especializados para el uso de valores de texto, también hemos usado modelos de machine learning más genéricos haciendo uso de TF-IDF para poder vectorizar el texto. Este tipo de vectorización, nos logra preservar datos como el número de apariciones de las palabras y las palabras del texto, sin embargo, perdemos información del contexto propio del texto lo cual puede afectar a la calidad de los modelos entrenados.

Para poder vectorizar el texto y luego hacer uso de este en nuestro modelo, hacemos uso de los pipelines de sklearn para poder vectorizar el texto y entrenar el modelo.

```

# Definir preprocesamiento del texto
preprocessor = ColumnTransformer(
    transformers=[
        # Texto: TF-IDF (título y autor por separado)
        ("text", TfidfVectorizer(max_features=1000), "text"),
        ("title", TfidfVectorizer(max_features=200), "title"),
    ],
    remainder="passthrough" # Mantener las columnas restantes sin cambios
)

# Pipeline completo: Preprocesamiento + Modelo
pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier", RandomForestClassifier()),
])

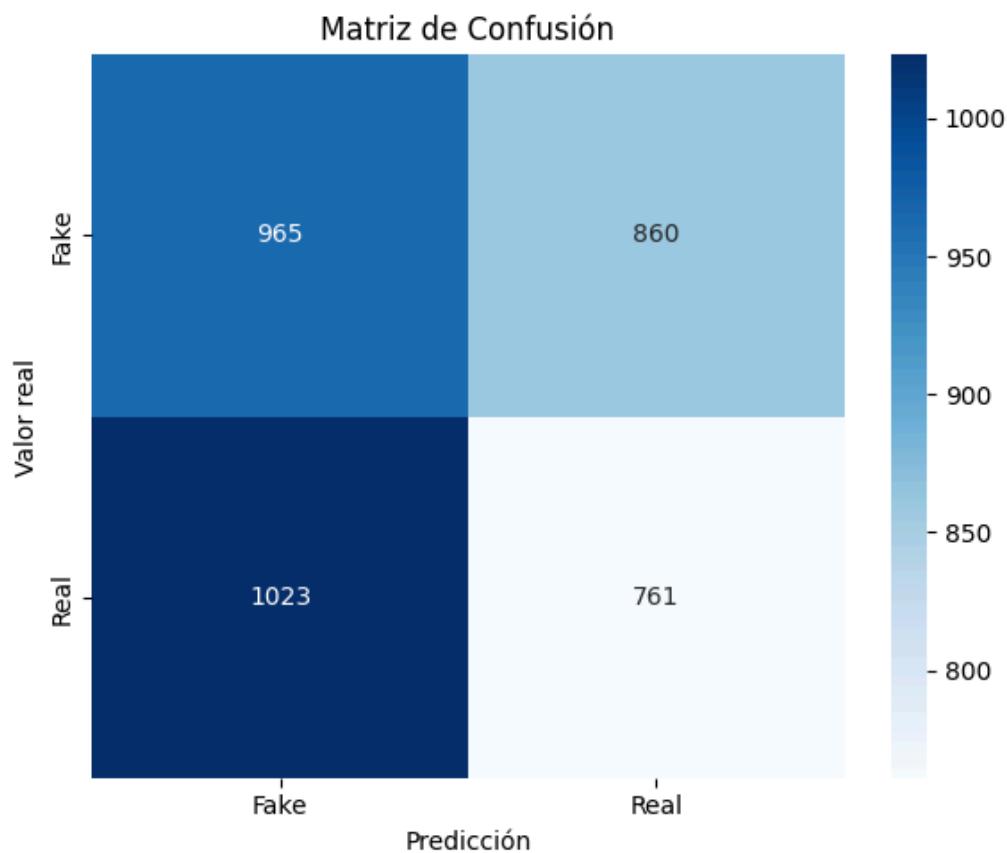
```

De nuevo para hacer la separación de los datos de entrenamiento y de prueba hacemos uso del `test_split` debido a la cantidad de datos y la distribución de las clases de forma pareja. Los modelos de machine learning supervisado que se han usado para estas pruebas han sido los siguientes: RandomForest, Gradient Boosted Decision Tree, Linear SVM y Perceptrón multicapa.

Random Forest

Para el random forest, se hizo uso de los datos estandarizados, y los resultados obtenidos fueron los siguientes:

Precision del modelo: 0.48				
Reporte de clasificación:				
	precision	recall	f1-score	support
0	0.49	0.53	0.51	1825
1	0.47	0.43	0.45	1784
accuracy			0.48	3609
macro avg	0.48	0.48	0.48	3609
weighted avg	0.48	0.48	0.48	3609

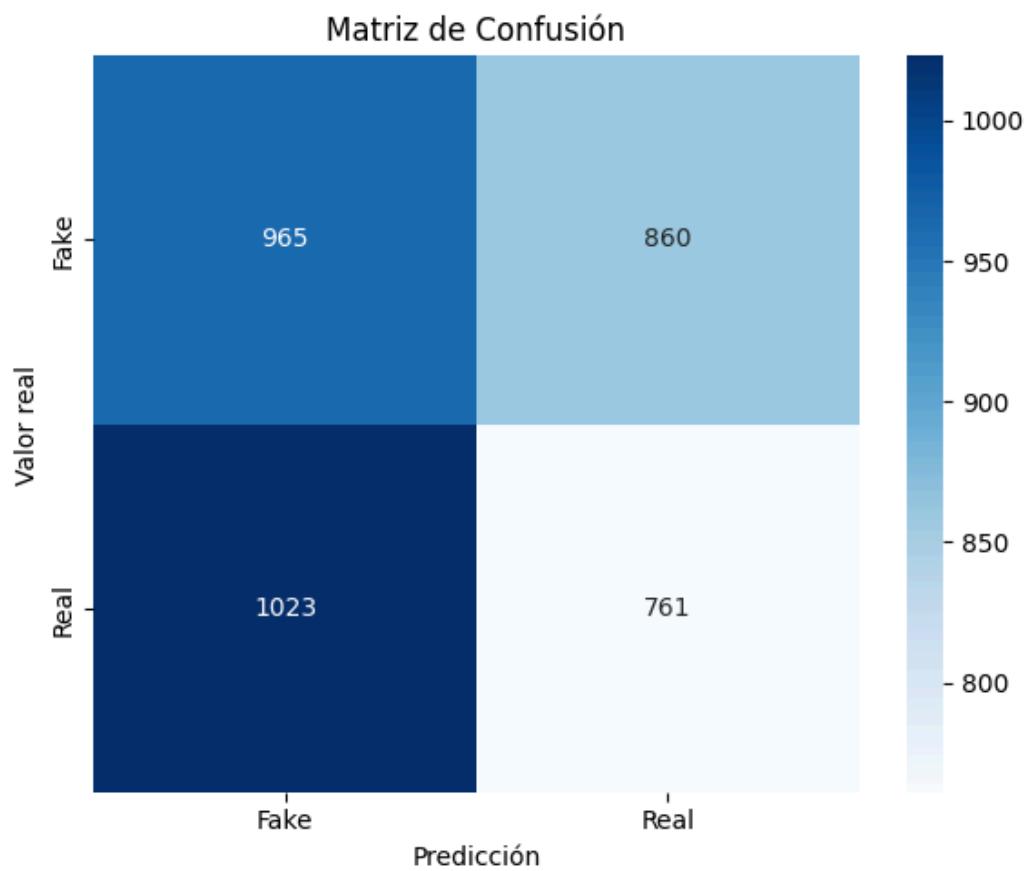


La precisión para ambas clases fue bastante pareja teniendo cierta tendencia a clasificar como fake las noticias. De todas formas los resultados no son satisfactorios, teniendo una precisión que no llega al 50%.

Gradient Boosted Decision Tree

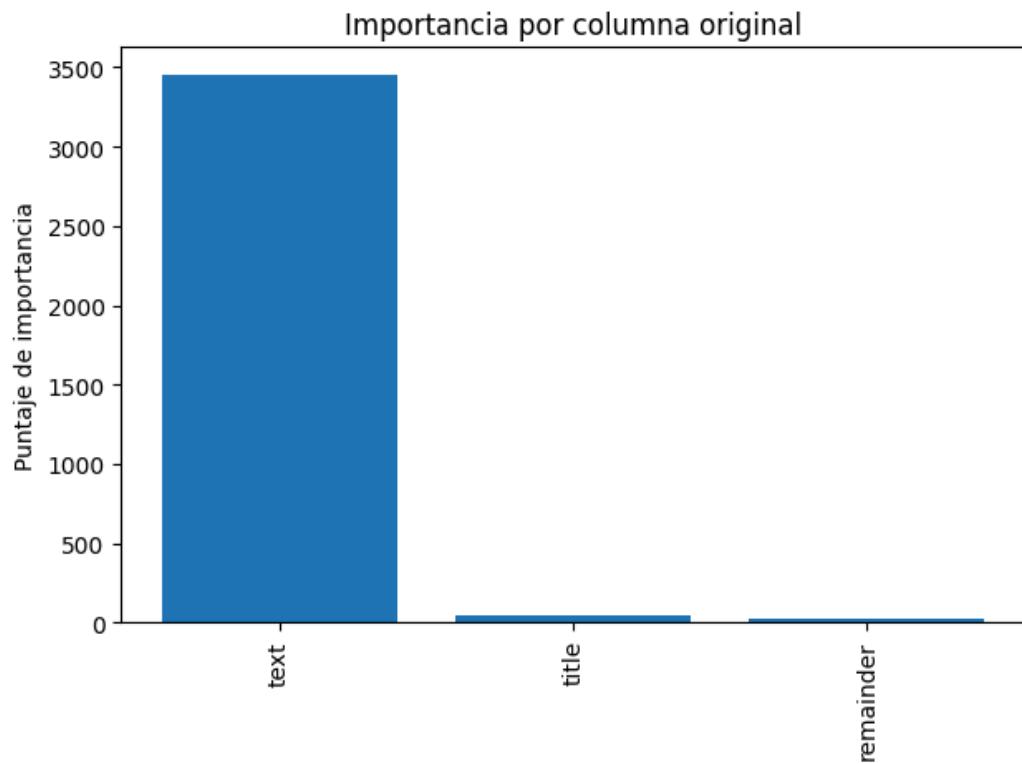
El segundo de los algoritmos que hemos usado es **XGBClassifier** de la librería de xgboost. Para este al igual que con el modelo anterior hemos usado los datos estandarizados, ya que no encontramos una gran diferencia entre usar los datos estandarizados o sin estandarización. Los resultados son los siguientes:

Precisión del modelo: 0.51				
Reporte de clasificación:				
	precision	recall	f1-score	support
0	0.51	0.51	0.51	1819
1	0.50	0.50	0.50	1790
accuracy			0.51	3609
macro avg	0.51	0.51	0.51	3609
weighted avg	0.51	0.51	0.51	3609



Podemos observar como los resultados son mejores que para el random forest, además este tiene una tendencia menor a clasificar como fake las noticias, siendo más equilibrado y logrando así para ambas clases una precisión de entorno al 50%, siguen siendo resultados malos pero sorprende que el árbol de gradientes se acerque a resultados de otros modelos enfocados al procesamiento de texto probados.

Además se aprovechó el uso de este modelo para obtener una gráfica de la importancia dada a los distintos datos del dataset.

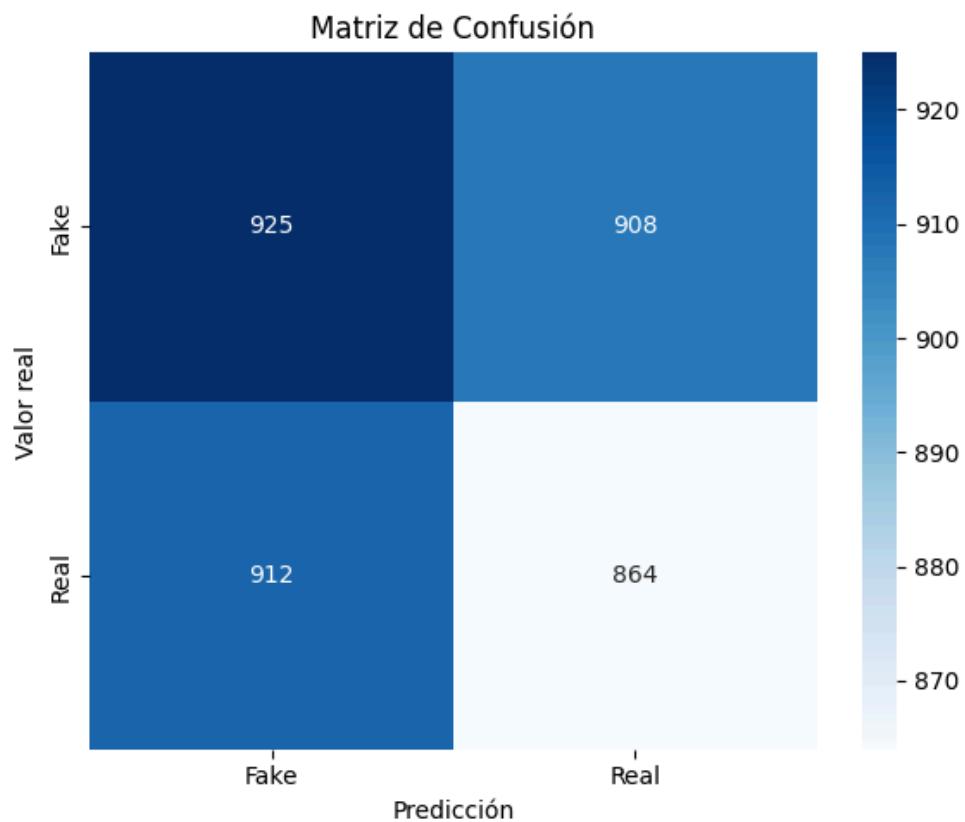


Se puede ver cómo para este, la importancia del campo de texto es mucho más importante que el resto, llegando a ser prácticamente el único dato que influye en la clasificación del modelo.

Linear SVM

El siguiente modelo usado es Linear SVM, esto debido a que los datos al ser vectorizados, corresponden a datos lineales y por tanto se entendió que este era el mejor kernel a usar para SVM. Además Linear SVM está optimizado para este kernel lo que lo hace más rápido y más optimizado que el modelo de SVM de sklearn. Se obtuvo el siguiente resultado:

Precisión del modelo: 0.50				
Reporte de clasificación:				
	precision	recall	f1-score	support
0	0.50	0.50	0.50	1833
1	0.49	0.49	0.49	1776
accuracy			0.50	3609
macro avg	0.50	0.50	0.50	3609
weighted avg	0.50	0.50	0.50	3609



Los resultados son muy semejantes a los del árbol de gradiente teniendo un 50% de precisión y una distribución bastante pareja de las clases en las que se ha clasificado.

Perceptrón multicapa

El último de los modelos que se ha usado es el perceptrón multicapa. Para dar con una combinación para este que nos dé el mejor resultado posible, se ha hecho uso de la herramienta oportuna. Esta permite para una serie de parámetros del perceptrón, definir una serie de valores sobre los que se generarán combinaciones aleatorias de esto y se analizará el resultado obtenido.

```

# Entrenamiento
x = df_standarized.drop(columns=["label"])
y = df_standarized[["label"]]
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

def objective(test):
    solver = test.suggest_categorical(
        "solver", ['adam', 'lbfgs', 'sgd'])
    activation= test.suggest_categorical(
        "activation", ['tanh', 'identity', 'logistic', 'relu'])
    first_layer_n_neurons = test.suggest_int(
        'first_layer_n_neurons', 10, 100, step=5)
    second_layer_n_neurons = test.suggest_int(
        'second_layer_n_neurons', 10, 100, step=5)
    hidden_layer_sizes = (first_layer_n_neurons, second_layer_n_neurons)
    learning_rate = test.suggest_categorical(
        'learning_rate', ['constant', 'invscaling', 'adaptive'])

    preprocessor = ColumnTransformer(
        transformers=[
            # Texto: TF-IDF (título y autor por separado)
            ("text", TfidfVectorizer(max_features=1000), "text"),
            ("title", TfidfVectorizer(max_features=200), "title"),
        ],
        remainder="passthrough" # Mantener las columnas restantes sin cambios
    )

```

Una vez definidos los parámetros a probar, se creó un estudio de optuna para comprobar cuál es la mejor de las combinaciones posibles.

```

study = optuna.create_study(directions=['maximize', 'maximize', 'minimize'])
study.optimize(objective, n_trials=100)

```

La combinación resultante fue la siguiente:

```

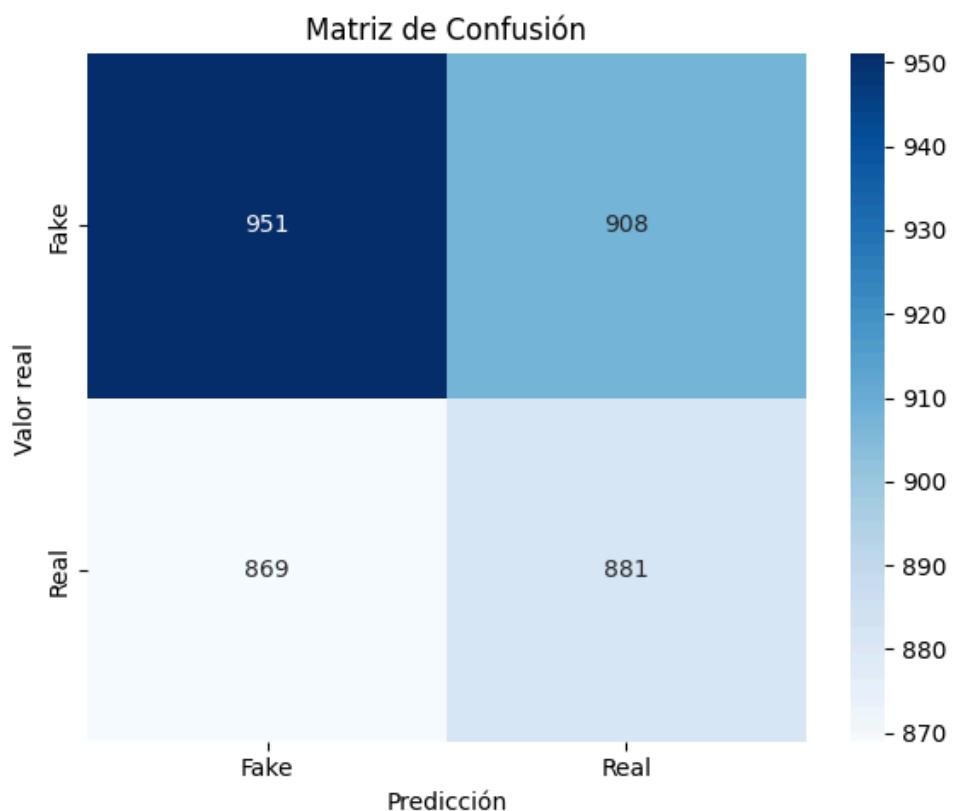
("classifier", MLPClassifier(
    learning_rate='invscaling',
    activation='relu',
    hidden_layer_sizes=(20, 35),
    solver='lbfgs',
    random_state=1234,
    max_iter=100)),
]

```

Una vez elegido cuál es la combinación de perceptrón multicapa que queremos usar, se pasó a la evaluación del modelo.

```
Precisión del modelo: 0.51
Reporte de clasificación:
      precision    recall   f1-score   support
0         0.52     0.51     0.52     1859
1         0.49     0.50     0.50     1750

accuracy                           0.51     3609
macro avg       0.51     0.51     0.51     3609
weighted avg    0.51     0.51     0.51     3609
```



Los resultados fueron parecidos a los del árbol de gradientes o de Linear SVC. En general para todos los modelos de Machine Learning, los resultados han sido muy semejantes. Estos resultados se mantienen parecidos entre los distintos modelos independientemente de cambios en algunos parámetros de estos o del uso o no de datos normalizados. Esto puede deberse a la alta importancia del texto en las predicciones como se ha visto con el árbol de gradientes, ya que esta importancia puede también estar afectando de igual manera al resto de modelos.

Por último y como nota final en lo que respecta a los modelos de Machine Learning supervisados que no pertenecen a los PLN. Se ha optado por el uso del dataset con datos normalizados ya que no se percibe apenas diferencia entre el uso de datos normalizados y datos sin normalizar para los modelos.

Conclusiones para los modelos IA estudiados

Una vez que hemos estudiado estos modelos, debemos elegir uno. Utilizaremos la matriz de dispersión y el classification report para evaluar y seleccionar.

Expliquemos la matriz de confusión. Tiene esta estructura:

	Predicción: Verdadero	Predicción: Falso
Real: Verdadero (TN)	Correcto (Verdadero Negativo)	Error (Falso positivo)
Real: Falso (FN)	Error (Falso Negativo)	Correcto (Verdadero Positivo)

En la siguiente tabla resumimos los resultados obtenidos en las matrices de dispersión calculadas:

Modelo	Correcto (Verdadero Negativo)	Error (Falso Negativo)	Error (Falso positivo)	Correcto (Verdadero Positivo)
RNN-LTSM. 80%-20%	0	0	1846	1763
RNN-LTSM. KFold, K=5	0	0	1809	1800
RNN-GRU.80 %-20%	0	0	1846	1763
RNN-GRU. KFold, K=5	0	0	1809	1800
BERT	0	0	1988	2012
Random Forest	761	860	1023	965
Gradient Boosted Decision Tree	888	883	902	936
Linear SVM	864	908	912	925
Perceptrón multicapa	881	908	869	951

Interpretamos estos valores.

En líneas generales, las redes neuronales recurrentes (LSTM y GRU) no tienen verdaderos negativos ni falsos negativos (TN y FN = 0). Esto sugiere que el modelo siempre predice “Falso” o “Verdadero”, pero nunca se equivoca en la clasificación de noticias verdaderas. Sin embargo, la alta cantidad de falsos positivos indica que el modelo está etiquetando muchas noticias reales como falsas.

Los modelos de aprendizaje automático clásico (Random Forest, Gradient Boosted Decision Tree, Linear SVM, etc.) tienen una mayor distribución de True Negative, False Negative, False Positive y True Positive. Esto implica que estos modelos tienen más equilibrio en su clasificación, en comparación con las redes neuronales. Sin embargo, algunos tienen valores significativos de falsos positivos, lo que significa que pueden estar dejando pasar noticias falsas como verdaderas.

Entrando en detalle de cada modelo:

- RNN-LSTM y RNN-GRU. Hay un alto número de verdaderos positivos, lo que indica buena detección de noticias falsas. Sin embargo, hay un alto número de falsos positivos, lo que significa que marcan muchas noticias verdaderas como falsas.
- BERT. Tiene la mayor cantidad de verdaderos positivos y también la mayor cantidad de falsos positivos. Si bien detecta bien las noticias falsas, también clasifica muchas verdaderas como falsas.
- Random Forest. Tiene varios verdaderos negativos, indicando que clasifica bien noticias reales. Sin embargo, hay un alto número de falsos negativos, lo que significa que deja pasar muchas noticias falsas.
- Gradient Boosted Decision Tree. Ofrece un buen balance entre todas las métricas, pero sigue teniendo un nivel moderado de falsos negativos y falsos positivos.
- Linear SVM y perceptrón multicapa. Ofrece una distribución de valores homogénea, aunque se dan falsos negativos y falsos positivos en cantidades similares, por lo que puede necesitar una optimización en el preprocesamiento.

Podemos considerar dos criterios para seleccionar un modelo apropiado:

- Priorizar la detección de noticias falsas. Los modelos BERT y las redes neuronales (LSTM/GRU) son las más apropiadas y, dentro de éstas dos, consideramos que BERT puede ser la más apropiada, por los valores obtenidos en la matriz de confusión (mayor cantidad de verdaderos positivos).
- Equilibrar la precisión entre noticias reales y falsas. En este caso, podemos considerar el modelo **Gradiente Boosted Decision Tree** como la mejor opción.

En la siguiente tabla, se resume los resultados obtenidos en los classification reports

Modelo	precisión	recall	f1-score	support
RNN-LTSM. 80%-20%	Fake: 0.0 Real: 0.49	Fake: 0.0 Real: 1.00	Fake: 0.0 Real: 0.66	Fake: 1846 Real: 1763
RNN-LTSM.	Fake:0.00	Fake: 0.0	Fake: 0.0	Fake: 1809

KFold	Real:0.50	Real: 1.0	Real: 0.67	Real: 1800
RNN-GRU.80 %-20%	Fake: 0.0 Real: 0.49	Fake: 0.0 Real: 1.00	Fake: 0.0 Real: 0.66	Fake: 1846 Real: 1763
RNN-GRU.KF old	Fake: 0.0 Real: 0.50	Fake: 0.0 Real: 1.00	Fake: 0.0 Real: 0.67	Fake: 1809 Real: 1800
BERT(solo texto)	Fake: 0.50 Real: 0.00	Fake: 1.0 Real: 0.00	Fake: 0.67 Real: 0.00	Fake: 2012 Real: 1988
BERT	Fake: 0.51 Real: 0.50	Fake: 0.60 Real: 0.45	Fake: 0.55 Real: 0.45	Fake: 1819 Real: 1790
Random Forest	Fake: 0.49 Real: 0.47	Fake: 0.53 Real: 0.43	Fake: 0.51 Real: 0.47	Fake: 1825 Real: 1784
Gradient Boosted Decision Tree	Fake: 0.51 Real: 0.50	Fake: 0.51 Real: 0.50	Fake: 0.51 Real: 0.50	Fake: 1819 Real: 1790
Linear SVM	Fake: 0.50 Real: 0.49	Fake: 0.50 Real: 0.49	Fake: 0.50 Real: 0.49	Fake: 1833 Real: 1776
Perceptrón multicapa	Fake: 0.52 Real: 0.49	Fake: 0.51 Real: 0.50	Fake: 0.52 Real: 0.50	Fake: 1859 Real: 1750

Con la información obtenida gracias a los classification report, podemos ver que hay un problema con las predicciones generadas por los modelos de redes neuronales recurrentes. Y es que estos modelos las predicciones tienden todas a predecir como real, es por esto por lo que no podemos recomendar ningún modelo de estos para nuestro caso de clasificación debido al alto sesgo que tiene hacia la clase de noticias reales. Esto lo podemos ver con las precisiones que tiene de 0.0 para la clase de Fake al igual que su recall y f1-score. Algo parecido sucede con BERT entrenado solo mediante texto, esté al contrario que como pasa con las recurrentes, esta tiende a predecir siempre falso, lo cual nos da una precisión, recall y f1-score de 0 para noticias verdaderas. Por esto se descarta el modelo de BERT entrenado solo con texto como un modelo a seleccionar.

Dentro del conjunto del resto de modelos, todos tienen valores bastante parecidos en lo que respecta a su clasificación. Si hay uno que podemos descartar a primera vista, sería random forest, ya que es de los modelos el que más tendencia tiene hacia una de las clases, siendo esta la de Real. Luego el resto de modelos tiende a tener una precisión parecida entre ambas clases siendo BERT la que más recall y f1-score tiene para las noticias reales, sacrificando en las falsas.

Estando todos los datos tan parejos entre el resto de modelos y no siendo ninguno bueno, los modelos con los que nos quedamos, serían BERT y Gradient Boosted Decision Tree. Esto debido a que los malos resultados de todos los modelos nos indican que los datos no eran los mejores para cumplir con el objetivo marcado. Por tanto a la hora de elegir un modelo hay que contar también con aquellos que tengan más potencial de mejora en la clasificación de noticias falsas. Y siendo BERT un

modelo basado en el procesamiento de texto, este es el que más potencial tiene de todos y sus resultados no se alejan del resto. Con respecto a Gradient Boosted Decision Tree, este modelo va muy bien para la clasificación de texto vectorizado además de ser eficiente. Pero si hay que quedarse con un modelo, este sería BERT por lo antes mencionado, siendo un modelo ideal para el procesamiento de texto y el cual guarda el contexto de las palabras, cosa que no pasa en modelos como Gradient Boosted Decision Tree.

Unificando los resultados y conclusiones de matriz de dispersión y clasificación report, el modelo IA a considerar sería el **transformer BERT**.

6. Aplicación de IA explicativa

Hemos considerado aplicar la IA explicativa para las Redes neuronales y los transformers.

Para las RNN-LSTM y RNN-GRU hemos considerado implementar las XAI SHAP (SHapley Additive Explanations) y LIME (Local Interpretable Model-Agnostic Explanations) y para Transformers, SHAP.

la XAI DICE hemos decidido no usarla porque al codificar el texto cada uno es una característica, lo cual hace que los contraejemplos son poco legibles.

Ofrecemos una breve explicación de ambas:

- SHAP: Nos proporciona valores de importancia para cada característica de entrada, indicando cómo contribuye a la decisión del modelo. Puede ser usado para entender cómo influyen las palabras del texto y los metadatos considerados (categorías, fuente, fecha) en la clasificación de noticias falsas.

SHAP es ideal para interpretar todo el modelo.

- LIME: Explica decisiones individuales del modelo entrenando un modelo interpretativo cercano a la predicción. Nos permite visualizar qué palabras o características influyen más en la decisión de una noticia falsa.

LIME nos ayuda a explicar casos individuales.

Inicialmente hemos programado SHAP y LIME para LSTM y GRU. A continuación, mostramos el código para LSTM y GRU.

LSTM

```
# Aplicación de SHAP para explicar el modelo LSTM
import shap
import lime
import lime.lime_tabular

# Crear un objeto SHAP explainer para LSTM
explainer_lstm = shap.Explainer(modelo, X_test)
shap_values_lstm = explainer_lstm(X_test)

# Visualización de características más importantes para LSTM
shap.summary_plot(shap_values_lstm, X_test, feature_names=tokenizer.index_word.values(), max_display=10)

# Aplicación de LIME para explicar el modelo LSTM
explainer_lime = lime.lime_tabular.limeTabularExplainer(training_data=X_train,
                                                          feature_names=news.columns[:-1], # Excluimos la columna de etiquetas
                                                          mode='classification')

idx = 10 # Índice de la instancia a explicar

# Explicación de una predicción con LSTM
exp_lstm = explainer_lime.explain_instance(X_test[idx], modelo.predict, num_features=5)
exp_lstm.show_in_notebook(show_table=True, show_all=False) # Mostrar la explicación en el cuaderno
```

GRU

```
# Aplicación de SHAP para explicar el modelo GRU
import shap
import lime
import lime.lime_tabular

# Crear un objeto SHAP explainer para GRU
explainer_gru = shap.Explainer(modelo, X_test)
shap_values_gru = explainer_gru(X_test)

# Visualización de características más importantes para GRU
shap.summary_plot(shap_values_gru, X_test, feature_names=tokenizer.index_word.values(), max_display=10)

# Aplicación de LIME para explicar el modelo GRU
explainer_lime = lime.lime_tabular.limeTabularExplainer(training_data=X_train,
                                                          feature_names=news.columns[:-1], # Excluimos la columna de etiquetas
                                                          mode='classification')

idx = 10 # Índice de la instancia a explicar

# Explicación de una predicción con GRU
exp_gru = explainer_lime.explain_instance(X_test[idx], modelo.predict, num_features=5)
exp_gru.show_in_notebook(show_table=True, show_all=False) # Mostrar la explicación en el cuaderno
```

⌚ 16m 18.4s
PermutationExplainer explainer: 0% | 13/3609 [14:03<72:31:35, 72.61s/it]

Sin embargo, hemos podido comprobar lo costoso que es, computacionalmente hablando.

Como vemos en la siguiente imagen, que corresponde a la ejecución del código para LSTM, tras 80 minutos de ejecución sólo hemos conseguido procesar un 3% del total de instancias.

Esto se debe a que ambas versiones calculan interpretaciones de cada instancia en el dataset. Por ello, optamos por optimizar su rendimiento. Tenemos dos opciones:

1. Reducir el tamaño de la muestra para SHAP y el número de características explicadas para LIME.
2. Para SHAP, reducir la complejidad del cálculo usando un método aproximado.
3. Para LIME, reducir el número de muestras generadas.

En la siguiente tabla resumimos la comparación de optimización de tiempos.

Técnica	Estrategia	Reducción de tiempo
SHAP	Usar una muestra más pequeña	Alta
SHAP	Algoritmo “permutation” en lugar de exacto	Media
LIME	Limitar el número de características	Alta
LIME	Reducir num_samples	Media

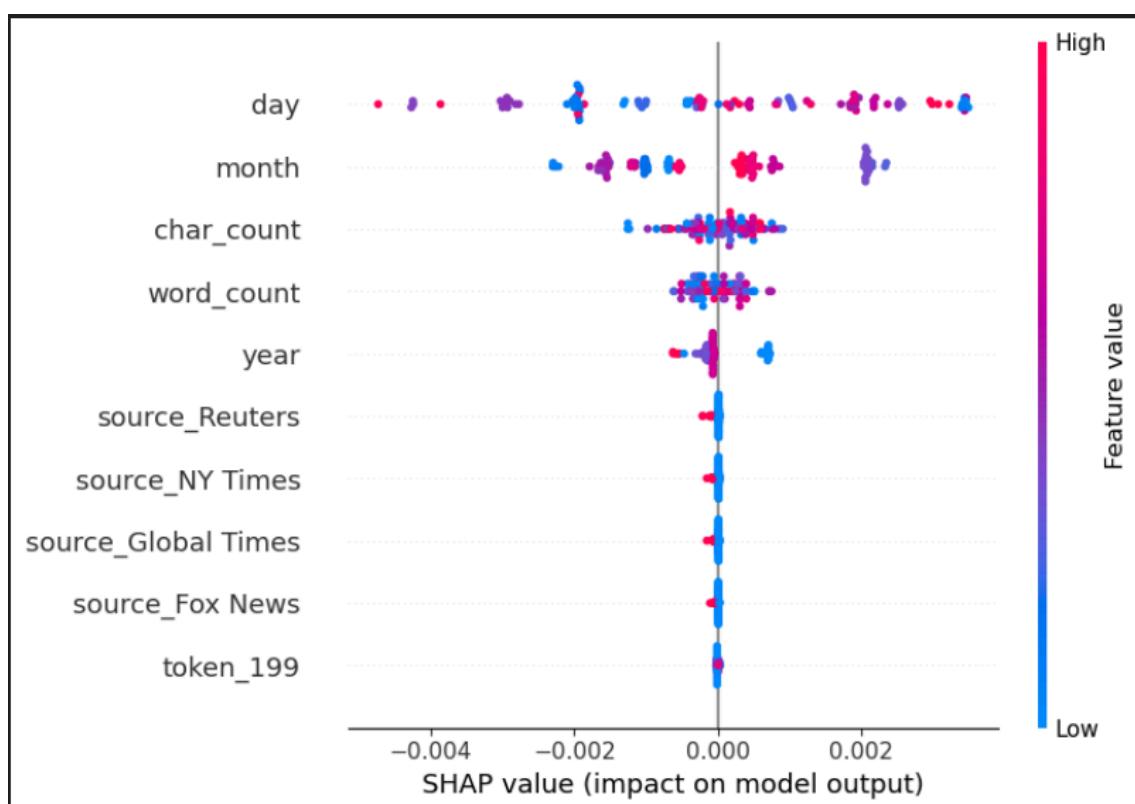
Optamos por reducir el tamaño de la muestra, ya que permite reducir el tiempo de ejecución significativamente.

Hemos considerado reducir la muestra a 100 ya que permite mejorar los tiempos de ejecución y aplicar la explicación enfocada a ejemplos clave. Podríamos tener menos generalización en el análisis, no capturándose correctamente patrones globales, así como producir un sesgo de los resultados si las 100 muestras no son representativas. Pero estamos en un entorno con limitación de tiempo y recursos computacionales. El incremento del tamaño de la muestra puede aplicarse como mejora del código desarrollado.

LSTM - SHAP y LIME

Tras 63 minutos, se ha completado la ejecución de SHAP y LIME para LSTM. Los resultados han sido los siguientes:

SHAP:



La característica “token_199” hace referencia a la palabra “impact”, como vemos en la siguiente ejecución:

```

palabra_asociada = [k for k,v in word_index.items() if v == 199]
if palabra_asociada:
    print(f"El token 199 corresponde a la palabra: {palabra_asociada[0]}")
✓ 0.0s

El token 143 corresponde a la palabra: hundred
El token 109 corresponde a la palabra: pull
El token 199 corresponde a la palabra: impact

```

Podemos ver como las características más interesantes para considerar una noticia como falsa o verdadera son:

- day
- month
- char_count
- word_count
- year

También, podríamos considerar que las fuentes tienen cierto peso, aunque no se ve ninguna que destaque significativamente, por lo podríamos ignorarlas.

Interpretando la gráfica, debemos explicar lo siguiente:

- Color rojo intenso → Valor alto de la característica.
- Color azul intenso → Valor bajo de la característica.
- Si los puntos rojos están más hacia la derecha, significa que esa característica aumenta la probabilidad de que una noticia sea falsa.
- Si los puntos azules están más hacia la izquierda, significa que esa característica reduce la probabilidad de que la noticia sea falsa.

Podemos ver que existe una fuerte relación en noticias falsas publicadas en unos meses concretos del año.

Con respecto a los días, podemos ver una relación entre días concretos en los que se generan noticias falsas. Aunque, al estar dispersos podemos considerar que el modelo no está considerando el día como una característica clave.

Sobre char_count, podemos ver que hay ligera mayor cantidad de puntos rojos hacia la derecha, lo que nos puede decir que el número de caracteres puede influir en que una noticia sea falsa, aunque no lo consideramos como algo tan claro como el caso de la característica mes, por ejemplo.

word_count, aunque parece que existe una distribución equilibrada entre los puntos distribuidos a la izquierda y la derecha, vemos una mayor tendencia de puntos rojos hacia la derecha, lo que nos indica que el número de palabras puede influir en la falsedad de una noticia.

Por último, year. Vemos que la mayor concentración de puntos están hacia la izquierda, pero cercanos a cero, por lo que podemos considerar que el año de publicación no tiene un gran efecto en la clasificación.

LIME:

Podemos ver cómo para un caso en concreto se puede clasificar una noticia y qué características se tienen en cuenta para dicha clasificación.

Para el modelo LSTM, vemos que la probabilidad de predecir es del 50% ‘Noticia Falsa’ y 50% ‘Noticia Verdadera’.

Las características que influyen en la falsedad de la noticia son:

- day, entre los días 8 y 16 del mes.
- token_143
- month, entre los meses 7 (julio) y 10 (octubre).

Mientras que las características que influyen en la veracidad de la noticia son:

- year, éste siendo menor o igual a 2023.
- token_109, siendo esto entre 434 y 532.



Podemos determinar qué palabras se corresponden con los token_143 y token_109. Desarrollamos y ejecutamos el siguiente código:

```
# A continuación, vamos a ver qué palabras se corresponden con el token 143 y 109.
word_index = tokenizer.word_index # Obtenemos el índice de las palabras

# Buscamos qué palabra corresponde al token 143
palabra_asociada = [k for k,v in word_index.items() if v == 143]
if palabra_asociada:
    print(f"El token 143 corresponde a la palabra: {palabra_asociada[0]}")

palabra_asociada = [k for k,v in word_index.items() if v == 109]
if palabra_asociada:
    print(f"El token 109 corresponde a la palabra: {palabra_asociada[0]}")
✓ 0.0s

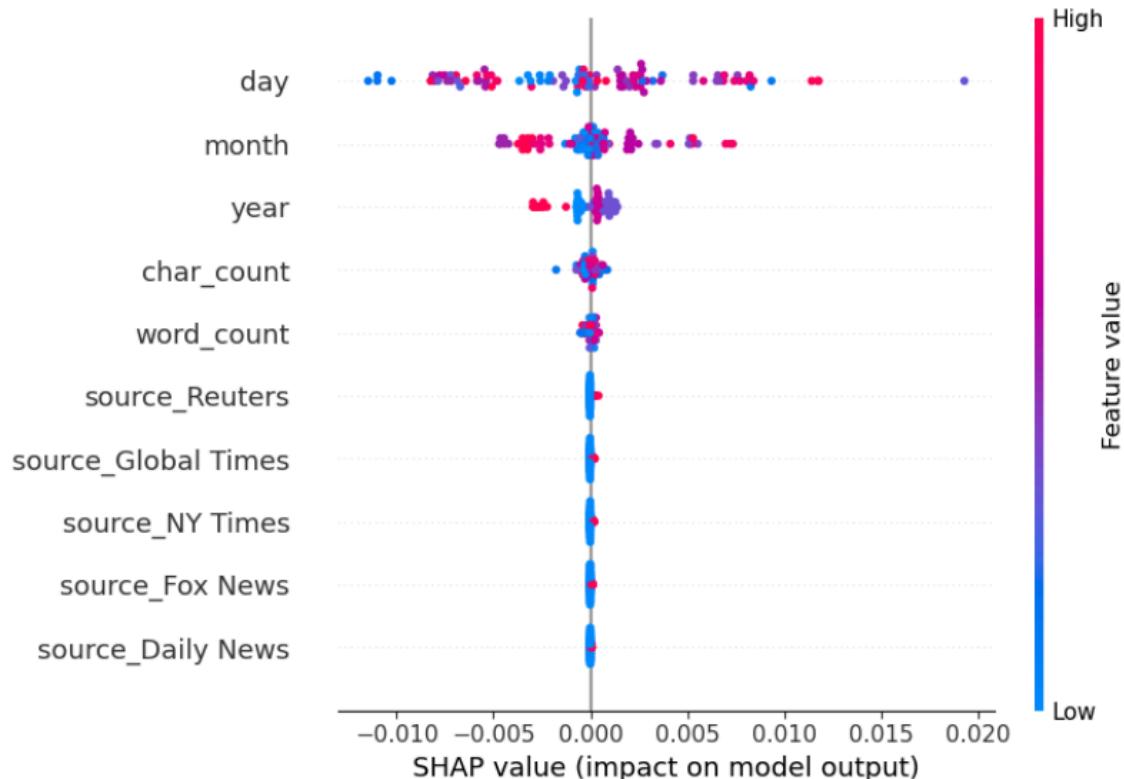
El token 143 corresponde a la palabra: hundred
El token 109 corresponde a la palabra: pull
```

Vemos que token_143 se corresponde con la palabra “hundred” y token_109 con “pull”.

GRU - SHAP y LIME

Tras 78 minutos, se ha completado la ejecución de SHAP y LIME para LSTM. Los resultados han sido los siguientes:

SHAP:



Para RNN-GRU, también podemos ver que las características más influyentes en clasificar una noticia como verdadera o falsa son:

- day
- month
- char_count
- word_count
- year

En este caso, vemos que las fuentes tienen el punto rojo ubicado a la derecha de la raya del cero, pero está prácticamente en cero. Así que también podríamos ignorarlas.

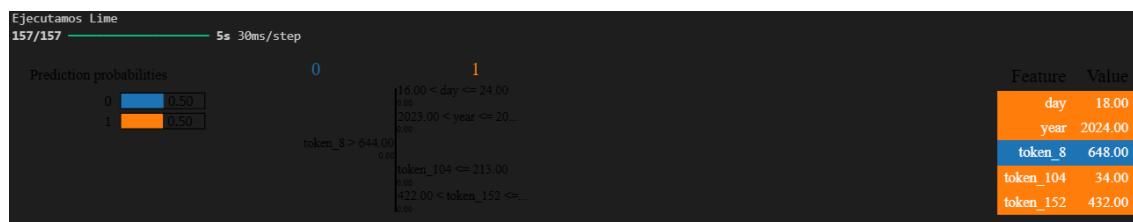
Las características month y days muestran valores contrarios a la obtenida para LSTM y SHAP. Esto es más hacia la izquierda que cero. Esto nos indica que noticias de esos días o meses tienen una menor probabilidad de ser falsas.

Sobre las características char_count y word_count, podemos ver que está todo distribuido homogéneamente en torno al cero, aunque hay valores por la izquierda y la derecha, con colores rojos. Esto es que, hay noticias en las que estas características influyen para que las noticias tengan cierta probabilidad de ser no falsas (puntos hacia la izquierda) y cierta probabilidad de ser falsas (puntos hacia la derecha).

Por último, year. Vemos una concentración de puntos rojos en la izquierda, lo que nos dice que para un año/años concretos, esas noticias tenían una baja probabilidad de ser falsas. También hay ciertos puntos en la derecha, próximos a cero y con un color entre azul y rojo, que indica que en esos años hubo noticias falsas, aunque no se puede determinar que fuera una tendencia en esos años.

Lime:

Con respecto a Lime, vemos que sólo considera la característica token_8 para clasificar una noticia como falsa, mientras que las características day, year, token_104 y token_152 para clasificar una noticia como verdadera.



Para obtener las palabras de los token_8, token_104 y token_152, ejecutamos el siguiente código. las palabras son:

- token_8: factor
- token_104: international
- token_152: enough

```
# A continuación, vamos a ver qué palabras se corresponden con los tokens 8, 104 y 152.
word_index = tokenizer.word_index # Obtenemos el índice de las palabras

# Buscamos qué palabra corresponde a los tokens 8, 104 y 152
palabra_asociada = [k for k,v in word_index.items() if v == 8]
if palabra_asociada:
    print(f"El token 8 corresponde a la palabra: {palabra_asociada[0]}")

palabra_asociada = [k for k,v in word_index.items() if v == 104]
if palabra_asociada:
    print(f"El token 104 corresponde a la palabra: {palabra_asociada[0]}")

palabra_asociada = [k for k,v in word_index.items() if v == 152]
if palabra_asociada:
    print(f"El token 152 corresponde a la palabra: {palabra_asociada[0]}")
✓ 0.0s

El token 8 corresponde a la palabra: factor
El token 104 corresponde a la palabra: international
El token 152 corresponde a la palabra: enough
```

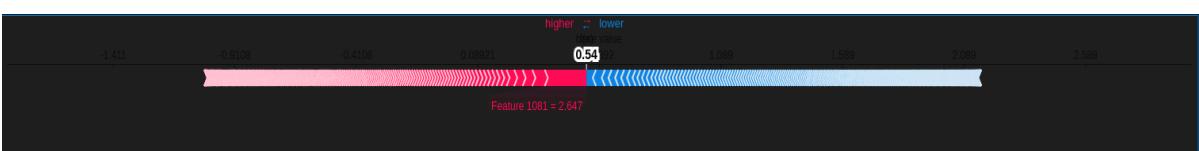
Vemos pocas características para considerar una noticia como falsa. Consideramos que los resultados obtenidos para LSTM-Lime eran más representativos.

Transformers

En lo que respecta al uso de XIA para los resultados obtenidos con transformers, para ser más concretos con BERT este no ha sido posible. Esto debido a que para lograr usar BERT en nuestro dataset, este se transforma en un dataset donde los datos del texto son codificados en tokens que luego se pasan por un tensor de torch para poder ser guardados en un dataset. Esto ha dificultado mucho la tarea de poder conseguir una explicación para este tipo de modelos.

Otra cosa a tener en cuenta y que hemos notado con las redes neuronales recursivas también, es que a la hora de codificar los campos de texto, estos pasan a ser transformados en tokens o features, lo que hace que las explicaciones que realizan tanto Lime como SHAP pasen a ser difíciles de entender y pierdan gran parte de su utilidad ya que necesitamos luego saber a qué palabra o texto corresponden los tokens de la explicación. Sin embargo estas explicaciones han sido útiles para ver como otros valores del dataset como el día, el año o el número de palabras han cobrado gran importancia en las predicciones obtenidas.

Por último hemos aplicado SHAP para explicar una predicción aleatoria del perceptrón multicapa y así ver qué valores son los que más influyen en sus predicciones.



Lo que hemos podido observar, es como para esta predicción, la mayoría de lo que se ha tenido en cuenta han sido features, es decir palabras de los campos de texto, lo que concuerda con las conclusiones sacadas gracias al árbol de gradientes de la importancia de el campo **text** frente al resto.

Como conclusión para ambos modelos, SHAP nos ofrece una información general para conocer la tendencia de nuestro modelo para un dataset concreto. Si queremos profundizar en casos individuales, podemos hacer zoom con Lime en RNN-LSTM/GRU.

7. Conclusiones

Tras completar los análisis previos, hemos podido determinar que el modelo IA **Transformers-BERT es el más idóneo** para la problemática cubierta por este trabajo.

En este trabajo hemos abordado los pasos seguidos en cada una de las prácticas de la asignatura para seleccionar, preparar y preprocesar los datasets a usar, definir los modelos de IA a utilizar, entenderlos y ajustarlos en función de nuestras necesidades, evaluar los resultados obtenidos, así como la bondad de éstos, explicándoles con XAI.

Además, hemos querido profundizar en redes neuronales vistas en las clases de teoría de la asignatura, con el objetivo de exponernos a éstas y poder incrementar nuestro conocimiento en la materia.

Este trabajo nos ha permitido ver lo complejo que pueden resultar problemas del tipo procesamiento de texto, análisis de sentimiento y veracidad del mismo y que se requiere de un procesamiento previo adecuado para poder trabajar con datos de calidad y que nos permita sacar conclusiones apropiadas. Puede aplicarse diferentes enfoques y centrar la atención y la ponderación en aspectos tales como fuente que publica una noticia, lo que requiere mayor conocimiento previo a aplicar para preprocesar el dataset a usar.

Como propuestas de mejora para una nueva versión de nuestro trabajo, diríamos el considerar la relación entre medio de publicación e histórico de noticias falsas identificadas, así como incluir la URL fuente para hacer consultas a herramientas que proporcionen el nivel de reputación. También, hemos identificado programas y herramientas, como [Google Fact Check Tools](#), que nos indica noticias falsas que han sido identificadas.