PROCESAMIENTO DEL LENGUAJE NATURAL

Word embeddings y *transformers* para clasificación de texto

Objetivos

Comparar diferentes modelos de clasificación de texto mediante técnicas basadas en word embeddings y transformers, adquiriendo la capacidad de aplicar un modelo neuronal para la clasificación de texto y comparar diferentes modelos entre sí, entendiendo los conceptos trabajados y realizando pequeñas experimentaciones para mejorar el notebook creado.

Jose Manuel Pinillos Rubio

Asignatura	Datos del alumno	Fecha
Procesamiento del Lenguaje Natural	Apellidos: Pinillos Rubio	31 de enero de 2025
	Nombre: Jose Manuel	

ÍNDICE

Asignatura	Datos del alumno	Fecha
Procesamiento del Lenguaje Natural	Apellidos: Pinillos Rubio	31 de enero de 2025
	Nombre: Jose Manuel	

Asignatura	Datos del alumno	Fecha
Procesamiento del Lenguaje Natural	Apellidos: Pinillos Rubio	31 de enero de 2025
	Nombre: Jose Manuel	

1. INTRODUCCIÓN

El **procesamiento del lenguaje natural (PLN)** ha evolucionado desde enfoques basados en reglas gramaticales hasta modelos avanzados de aprendizaje profundo. Inicialmente, se utilizaban técnicas de **análisis morfosintáctico** y **gramáticas libres de contexto** para estructurar el lenguaje, pero estos métodos eran insuficientes para manejar la ambigüedad y complejidad del texto. Con la llegada del **aprendizaje automático**, las redes neuronales clásicas mejoraron la clasificación de textos al aprender representaciones densas de palabras, aunque tenían dificultades para capturar dependencias a largo plazo. Para superar estas limitaciones, surgieron las **redes recurrentes (RNN)** y las **redes convolucionales (CNN)**, que lograron un mejor procesamiento de secuencias. Finalmente, los **Transformers** revolucionaron el PLN gracias a su mecanismo de atención, que permite analizar el contexto global de una oración sin depender de procesamiento secuencial, mejorando la precisión y la capacidad de generalización del modelo.

Este trabajo desarrolla un análisis detallado sobre el entrenamiento y evaluación de modelos de clasificación de texto, comparando una **red neuronal clásica** con un **modelo basado en Transformers**. El objetivo es comprender cómo estas arquitecturas procesan y clasifican textos pertenecientes a diferentes categorías, utilizando un conjunto de datos estructurado y técnicas de procesamiento del lenguaje natural.

El estudio abarca la preparación y preprocesamiento de los datos, incluyendo la tokenización con *spaCy*, la eliminación de información irrelevante y la división del conjunto de datos en entrenamiento y validación. Posteriormente, se construyen y entrenan ambos modelos, analizando sus diferencias en términos de arquitectura, precisión y capacidad de generalización. A través de la evaluación con textos de prueba, se observa cómo cada modelo clasifica diferentes temas y se comparan sus fortalezas y limitaciones.

Finalmente, se identifican posibles mejoras y desafíos en la clasificación de textos, abordando aspectos como el sobreajuste, la sensibilidad a la estructura del lenguaje y la posibilidad de adaptar los modelos a otros idiomas. Este análisis permite evaluar la efectividad de cada enfoque y resalta la importancia de seleccionar la arquitectura adecuada según el contexto y los datos disponibles.

Asignatura	Datos del alumno	Fecha
Procesamiento del Lenguaje Natural	Apellidos: Pinillos Rubio	31 de enero de 2025
	Nombre: Jose Manuel	

2. PREGUNTAS

2.1 Pregunta 1

2.1.1 Utilizando el tokenizador de spacy, que ya conoces, calcula el número promedio de tokens de una muestra de 15 ficheros de la categoría «com.graphics».

```
# Importamos la biblioteca spaCy para el procesamiento del lenguaje
natural
import spacy
import en_core_web_sm  # Cargamos el modelo de spaCy para el idioma
inglés

# Cargamos el modelo de procesamiento de texto en inglés
"en_core_web_sm"
nlp = en_core_web_sm.load()

# Inicializamos una variable para contar el número total de tokens
total_tokens = 0

# Aseguramos que tomamos los primeros 15 archivos
num_files = 15
fnames_subset = fnames[:num_files]  # Tomamos los primeros 15 archivos
de la lista

# Iteramos sobre los primeros 15 archivos de la categoría
"comp.graphics"
for fname in fnames_subset:
    # Leemos el contenido del archivo y lo procesamos con spaCy
    doc = nlp(pathlib.Path(data_dir / "comp.graphics" /
fname).read_text(encoding="latin-1"))

    # Obtenemos la longitud del documento en tokens y la sumamos al
total
    total_tokens += len(doc)

# Calculamos el número promedio de tokens en los 15 archivos analizados
average_tokens = total_tokens / num_files

# Imprimimos el resultado
print(f"Promedio de tokens en los primeros 15 documentos de
'comp.graphics': {average_tokens:.2f}")
```

Este código calcula el número promedio de tokens en 15 archivos de la categoría *comp.graphics* utilizando *spaCy* para la tokenización. Para ello, primero se inicializa la variable `total_tokens` en cero, que servirá para almacenar la cantidad total de tokens en todos los archivos procesados.

Se define la variable `num_files` con el valor 15 para indicar cuántos archivos se van a analizar. Luego, se obtiene una lista con los nombres de los primeros 15 archivos dentro de la carpeta *comp.graphics* utilizando `fnames[:num_files]`, lo que garantiza que solo se procesen los archivos necesarios.

A continuación, se inicia un bucle que recorre cada uno de estos archivos. Para cada archivo, se construye su ruta completa utilizando `pathlib.Path(data_dir / "comp.graphics" / fname)`, asegurando así que se accede correctamente a su contenido. La función `read_text(encoding="latin-1")` se encarga de leer el contenido del archivo como texto, utilizando la codificación *latin-1* para manejar caracteres especiales sin errores. Una vez obtenido el texto del archivo, se pasa a la función `nlp()` de spaCy, que tokeniza el texto y devuelve un objeto `doc` que contiene los *tokens* generados.

La cantidad de tokens en el documento se obtiene con `len(doc)`, que devuelve el número total de *tokens* en el texto procesado. Este valor se suma a la variable `total_tokens`, acumulando así el número total de *tokens* de todos los archivos analizados.

Después de recorrer los 15 archivos y sumar la cantidad total de *tokens*, se calcula el promedio dividiendo `total_tokens` entre `num_files`. Finalmente, el resultado se imprime en pantalla con un mensaje claro, utilizando una *f-string* para formatear la salida y asegurando que el promedio de *tokens* se muestre con solo dos decimales mediante `{average_tokens:.2f}`.

El resultado obtenido indica que, en promedio, cada uno de los primeros 15 documentos de la categoría *comp.graphics* contiene aproximadamente **276.13 tokens**. Esto refleja el tamaño medio de los textos en esta categoría después del proceso de tokenización con *spaCy*.

2.2 Pregunta 2

El código proporcionado lee los ficheros uno a uno y, antes de generar el catálogo de datos de entrenamiento y validación, descarta las diez primeras líneas de cada fichero.

2.2.1 ¿Cuál es el trozo de código en el que se realiza dicho descarte?

En el código anterior, dentro del bucle `for`, hay una sección que se encarga de eliminar las diez primeras líneas de cada archivo antes de almacenarlo en la lista de muestras. Este procesamiento se realiza en el siguiente fragmento:

```
# Eliminamos las primeras 10 líneas del contenido (posiblemente
metadatos o cabecera)
lines = content.split("\n")  # Dividimos el texto en líneas
lines = lines[10:]  # Eliminamos las primeras 10 líneas
content = "\n".join(lines)  # Volvemos a unir las líneas en un solo
texto
```

Este fragmento de código realiza un preprocesamiento del contenido de cada archivo. Primero, la función `content.split("\n")` divide el texto en una lista de líneas, separando cada fragmento de texto según los saltos de línea. Luego, la instrucción `lines = lines[10:]` descarta las primeras diez líneas de la lista, eliminando información que suele corresponder a metadatos, encabezados de correos electrónicos o referencias internas dentro del conjunto de datos. Finalmente, `content = "\n".join(lines)` reconstruye el texto uniendo nuevamente las líneas restantes, asegurando que la estructura del contenido se mantenga, pero sin los elementos iniciales considerados innecesarios para el análisis.

2.2.2 ¿Por qué crees que se descartan dichas líneas?

Las primeras diez líneas se descartan porque contienen principalmente metadatos, como referencias a otros mensajes, rutas de servidores, remitentes, organizaciones y encabezados de correo electrónico. Estos elementos no forman parte del contenido principal del mensaje y podrían introducir ruido o sesgos en el modelo de clasificación. Al eliminarlas, se mantiene solo el cuerpo del texto, asegurando que la clasificación se base en la información relevante de cada documento.

2.2.3 ¿Por qué diez y no otro número?

Se eliminan exactamente diez líneas porque, al analizar previamente el contenido de los archivos con el siguiente código:

```
# Ejemplo de un texto de la categoría "talk.politics.misc"
print(open(data_dir / "talk.politics.misc" / "178463").read())
```

Se observa que las primeras diez líneas contienen información estructural que no es relevante para la clasificación del texto. En este ejemplo, se incluyen encabezados como `Xref`, `Newsgroups`, `Path`, `From`, `Subject`, `Message-ID`, `Lines`, `Sender`, `Organization`, `References` y `Date`. Estos campos corresponden a metadatos del sistema de *newsgroups* que identifican la procedencia del mensaje, el remitente y referencias a otros mensajes, pero no aportan información significativa para la tarea de clasificación.

El número diez no es arbitrario, sino que se basa en la estructura observada en estos archivos. Al eliminar exactamente estas diez líneas, se garantiza que el modelo trabaje únicamente con el contenido real del mensaje sin interferencias de información técnica o administrativa. Esto permite que el texto procesado represente mejor la categoría a la que pertenece sin incluir datos que podrían introducir ruido o sesgos en la clasificación.

2.3 Pregunta 3

2.3.1 ¿Qué se controla con el parámetro `validation_split`?

En el código anterior, el parámetro `validation_split` define el porcentaje de datos que se utilizarán para la validación del modelo. En este caso, se ha establecido en 0.2, lo que significa que el 20% de las muestras se reserva para validación y el 80% restante se utiliza para el entrenamiento.

```
# Definimos el porcentaje de datos que se usará para validación
validation_split = 0.2  # 20% de los datos serán utilizados para
validación
```

2.3.2 ¿Por qué se ha elegido ese valor?

Este valor ha sido elegido para disponer de un conjunto de validación suficientemente representativo sin reducir en exceso la cantidad de datos destinados al entrenamiento, lo que permite evaluar el rendimiento del modelo de manera equilibrada.

2.3.3 ¿qué ocurre si lo modificas?

Si se modifica este valor, el tamaño relativo de los conjuntos de entrenamiento y validación cambiará. Un valor más alto reduciría el número de muestras de entrenamiento, lo que podría afectar la capacidad del modelo para generalizar. Por el contrario, un valor más bajo dejaría menos datos para validación, lo que podría dificultar la evaluación del rendimiento del modelo y aumentar el riesgo de sobreajuste si la validación no es suficientemente representativa.

2.4 Pregunta 4

2.4.1 Imprime por pantalla un ejemplo (es decir, un elemento del array) de `train_samples`, `val_samples`, `train_labels` y `val_labels`. A tenor de las etiquetas que se utilizan.

```
import random # Importamos la librería random para generar números aleatorios

# Establecemos una semilla fija para obtener siempre el mismo número aleatorio
random.seed(13)

# Generamos un índice aleatorio reproducible para los datos de entrenamiento
random_index_train = random.randint(0, len(train_samples) - 1)

# Generamos un índice aleatorio reproducible para los datos de validación
random_index_samples = random.randint(0, len(val_samples) - 1)

# Imprimimos un ejemplo aleatorio fijo del conjunto de entrenamiento y su etiqueta correspondiente
print("Ejemplo aleatorio fijo de train_samples:")
print(train_samples[random_index_train])
print("Etiqueta correspondiente en train_labels:", train_labels[random_index_train])

# Imprimimos un ejemplo de los datos de validación y su etiqueta correspondiente
print("\nEjemplo de val_samples:")
print(val_samples[random_index_samples])
print("Etiqueta correspondiente en val_labels:", val_labels[random_index_samples])
```

Este código selecciona e imprime un ejemplo aleatorio tanto del conjunto de entrenamiento como del conjunto de validación, asegurando que la selección sea reproducible.

Para lograrlo, primero se importa la librería `random`, que permite generar números aleatorios. Luego, se fija una semilla con `random.seed(13)`, lo que garantiza que cada vez que se ejecute el código, se generarán los mismos valores aleatorios.

A continuación, se generan dos índices aleatorios dentro del rango válido de cada conjunto de datos. `random.randint(0, len(train_samples) - 1)` selecciona un índice aleatorio dentro del conjunto de entrenamiento, mientras que `random.randint(0, len(val_samples) - 1)` hace lo mismo para el conjunto de validación.

Finalmente, se imprimen los ejemplos seleccionados junto con sus etiquetas correspondientes. `train_samples[random_index_train]` muestra un documento aleatorio del conjunto de entrenamiento, mientras que `val_samples[random_index_samples]` muestra un documento aleatorio del conjunto de validación. En ambos casos, también se imprime la etiqueta correspondiente para identificar la categoría a la que pertenece el texto seleccionado.

2.4.2 ¿Qué tarea crees que se está intentando entrenar?

El código está preparando los datos para entrenar un **modelo de clasificación de texto**. Se está estructurando un conjunto de datos en el que cada muestra es un documento de texto y cada documento tiene una etiqueta numérica correspondiente a una categoría específica dentro del conjunto de datos *20 Newsgroups*.

Dado que los datos provienen de diferentes categorías de newsgroups, la tarea de aprendizaje automático que se intenta entrenar es un **modelo de clasificación de textos en múltiples categorías** (*multi-class text classification*). El objetivo del modelo será aprender a asociar nuevos textos con la categoría correcta basándose en su contenido.

2.5 Pregunta 5

2.5.1 Con `output_sequence_length` se establece un tamaño fijo para la salida de Vectorizer. ¿Por qué se necesita un tamaño fijo y por qué se ha elegido el valor 200?

En las redes neuronales, el tamaño de la entrada debe ser constante para que el modelo pueda procesar los datos de manera uniforme. Por esta razón, `output_sequence_length` se establece con un valor fijo, en este caso 200, asegurando que todas las secuencias tengan la misma longitud antes de ser introducidas en la red neuronal.

El valor 200 ha sido elegido porque la arquitectura del modelo ha sido diseñada para trabajar con entradas de esta dimensión, lo que significa que la capa de entrada de la red neuronal está configurada para recibir vectores de tamaño 200. Si un documento tiene menos de 200 tokens, se aplicará *padding* para completar la secuencia hasta la longitud deseada, mientras que si excede este límite, se truncará. De esta manera, se garantiza que todos los ejemplos tengan una representación uniforme, lo que facilita el procesamiento y la eficiencia del modelo durante el entrenamiento y la inferencia.

2.6 Pregunta 6

2.6.1 Indica cuál es la precisión del modelo en el conjunto de datos de entrenamiento y en el conjunto de datos de validación.

2.6.1.1 Precisión de la red neuronal clásica en entrenamiento y validación.

```
# Evaluamos la precisión del modelo clásico en el conjunto de
entrenamiento
train_loss_clasico, train_acc_clasico =
modeloClasico.evaluate(x_train, y_train)

# Evaluamos la precisión del modelo clásico en el conjunto de
validación
val_loss_clasico, val_acc_clasico = modeloClasico.evaluate(x_val,
y_val)

# Imprimimos los resultados de precisión para el modelo clásico
print(f"Precisión del modelo clásico en entrenamiento:
{train_acc_clasico:.4f}")
print(f"Precisión del modelo clásico en validación:
{val_acc_clasico:.4f}")
```

```
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - acc: 0.9706 - loss: 0.0660
94/94 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - acc: 0.6751 - loss: 1.2803
Precisión del modelo clásico en entrenamiento: 0.9719
Precisión del modelo clásico en validación: 0.6852
```

Este código evalúa el rendimiento de la **red neuronal clásica** en los conjuntos de entrenamiento y validación. Se utiliza `evaluate()` para calcular la **pérdida y la precisión** del modelo en cada conjunto de datos.

Primero, `modeloClasico.evaluate(x_train, y_train)` obtiene la precisión en el conjunto de entrenamiento, indicando qué tan bien el modelo ha memorizado los datos en los que ha sido entrenado. Luego, `modeloClasico.evaluate(x_val, y_val)` mide su desempeño en el conjunto de validación, reflejando su capacidad para generalizar a datos no vistos.

Finalmente se imprimen los resultados de precisión en ambos conjuntos.

2.6.1.2 Precisión del modelo basado en Transformers en entrenamiento y validación.

```
# Evaluamos la precisión del modelo clásico en el conjunto de
entrenamiento
train_loss_clasico, train_acc_clasico =
modeloClasico.evaluate(x_train, y_train)

# Evaluamos la precisión del modelo clásico en el conjunto de
validación
val_loss_clasico, val_acc_clasico = modeloClasico.evaluate(x_val,
y_val)

# Imprimimos los resultados de precisión para el modelo clásico
print(f"Precisión del modelo clásico en entrenamiento:
{train_acc_clasico:.4f}")
print(f"Precisión del modelo clásico en validación:
{val_acc_clasico:.4f}")
```

```
375/375 ━━━━━━━━━━━━━━━━━━━━ 15s 39ms/step - acc: 0.9720 - loss: 0.0578
94/94 ━━━━━━━━━━━━━━━━━━━━ 3s 34ms/step - acc: 0.8255 - loss: 0.9343
Precisión del modelo Transformers en entrenamiento: 0.9724
Precisión del modelo Transformers en validación: 0.8366
```

Este código evalúa la **precisión del modelo basado en Transformers** en los conjuntos de entrenamiento y validación. Se sigue el mismo procedimiento que en la red neuronal clásica, pero ahora aplicando `evaluate()` sobre `modeloTransformers`.

Primero, `modeloTransformers.evaluate(x_train, y_train)` mide qué tan bien el modelo ha aprendido sobre los datos de entrenamiento. Luego, `modeloTransformers.evaluate(x_val, y_val)` analiza su rendimiento en datos de validación, verificando su capacidad para generalizar a ejemplos nuevos.

Finalmente se imprimen los resultados de precisión en ambos conjuntos.

2.6.2 ¿Qué interpretación puedes dar? Haz un análisis comparativo de los dos modelos ejecutados.

Los resultados obtenidos muestran que ambos modelos tienen un rendimiento muy similar en el conjunto de entrenamiento, alcanzando precisiones cercanas al 97%. Esto indica que los dos han logrado aprender los patrones presentes en los datos de manera eficiente y han

ajustado sus parámetros para minimizar el error en este conjunto. Sin embargo, la diferencia más relevante aparece en el conjunto de validación, donde el modelo basado en Transformers logra un 81.93% de precisión, mientras que la red neuronal clásica solo alcanza un 67.69%. Esta discrepancia sugiere que el modelo clásico está significativamente más sobreajustado, ya que su desempeño en datos no vistos es mucho peor en comparación con lo que logra en entrenamiento.

El sobreajuste en la red neuronal clásica se debe probablemente a la forma en que procesa la información. Al no contar con mecanismos avanzados de captura de contexto, como la autoatención de los Transformers, su aprendizaje depende más de la frecuencia de palabras individuales y patrones específicos en los datos de entrenamiento, lo que reduce su capacidad de generalización. Como resultado, aunque logra una alta precisión en entrenamiento, su rendimiento cae drásticamente en validación, lo que indica que ha memorizado los datos en lugar de extraer patrones generales.

Por otro lado, el modelo basado en Transformers muestra una diferencia menor entre la precisión en entrenamiento y validación, lo que demuestra que ha logrado generalizar mejor los datos aprendidos. Su capacidad para capturar relaciones entre palabras mediante el mecanismo de atención multi-cabeza le permite analizar mejor la semántica del texto, en lugar de depender solo de la frecuencia de aparición de palabras. Aun así, la diferencia de aproximadamente 15 puntos porcentuales entre el conjunto de entrenamiento y validación sugiere que, aunque menor que en el modelo clásico, todavía hay cierto grado de sobreajuste. Esto podría mejorarse con técnicas adicionales de regularización o un ajuste más preciso de los hiperparámetros.

En general, estos resultados refuerzan la superioridad del enfoque basado en Transformers en tareas de clasificación de texto. Mientras que la red neuronal clásica muestra una gran caída de rendimiento al enfrentarse a datos nuevos, el modelo basado en Transformers demuestra una mayor capacidad para generalizar, evitando en gran medida el sobreajuste y obteniendo predicciones más precisas en validación. Esto confirma que el uso de mecanismos de atención proporciona una ventaja clara al modelar relaciones más complejas dentro del texto, lo que permite al modelo aprender representaciones más ricas y adaptarse mejor a datos desconocidos.

2.7 Pregunta 7

2.7.1 En la parte final del código se hace un análisis cualitativo de la salida. Explica el funcionamiento de este análisis e interpreta los resultados. Haz también, en este punto, un análisis comparativo de los dos modelos ejecutados.

En la parte final del código, se realizó un análisis cualitativo de la salida de los modelos probando su capacidad de clasificación con distintos textos de ejemplo. En cada caso, se introdujo una frase representativa de una temática específica, como **gráficos por computadora**, **política** y **religión**, y se analizó la categoría asignada por el modelo. El proceso consistió en convertir el texto en un tensor compatible, aplicar la vectorización para transformarlo en una secuencia de tokens numéricos y luego pasarlo al modelo entrenado para obtener una predicción. Finalmente, se extrajo la categoría con mayor probabilidad y se comparó con la temática esperada.

Los resultados muestran diferencias en la precisión y la capacidad de generalización entre los modelos. En el caso de un texto sobre gráficos por computadora, ambos modelos lograron clasificarlo correctamente en la categoría `comp.graphics`, lo que indica que pudieron identificar términos clave como *computer graphics* y *3D modeling*. Esto sugiere que las características de esta categoría han sido bien aprendidas por ambas arquitecturas, permitiendo una clasificación precisa en este caso.

Para un texto relacionado con política y legislación, los dos modelos lo clasificaron en `talk.politics.guns`, a pesar de que el contenido hablaba sobre tribunales, leyes y políticos. Esto sugiere que en el conjunto de datos existe una fuerte correlación entre estos términos y el debate sobre armas, lo que llevó a ambos modelos a realizar la misma clasificación. Este resultado refuerza la idea de que los modelos aprenden patrones a partir de las asociaciones más frecuentes en los datos de entrenamiento, lo que puede influir en ciertas predicciones.

La diferencia más significativa se observa en la clasificación de un texto sobre religión. Mientras que la red neuronal clásica lo clasificó erróneamente dentro de `rec.sport.baseball`, el modelo basado en Transformers lo asignó a `alt.atheism`, lo que resulta más coherente con el significado del texto. Este resultado indica que la arquitectura basada en Transformers ha captado mejor las relaciones semánticas dentro del

texto y ha logrado una clasificación más precisa. La red neuronal clásica, en cambio, parece haber asignado un peso mayor a ciertas palabras sin considerar el contexto completo, lo que llevó a una predicción errónea.

Este análisis cualitativo confirma que, si bien ambos modelos pueden clasificar correctamente ciertos textos, el modelo basado en Transformers demuestra una mayor capacidad para comprender el contexto y generalizar mejor, especialmente en textos más ambiguos. Esto se debe a su capacidad para analizar la relación entre palabras mediante mecanismos de atención, lo que le permite diferenciar mejor los significados y evitar errores de clasificación más drásticos como los observados en el modelo clásico.

2.8 Pregunta 8

2.8.1 Explica algunas de las limitaciones que puedes encontrar al modelo entrenado.

El modelo entrenado, tanto en su versión clásica como en la basada en **Transformers**, presenta diversas limitaciones que pueden afectar su rendimiento y capacidad de generalización. Una de las principales limitaciones es la **dependencia del conjunto de datos de entrenamiento**. Si el *dataset* contiene sesgos en la distribución de las clases o en la forma en que se presentan ciertos temas, el modelo puede aprender estas tendencias y reflejarlas en sus predicciones, lo que puede llevar a clasificaciones incorrectas en textos que no sigan los patrones dominantes del conjunto de datos.

Otra limitación importante es la **sensibilidad a la redacción y vocabulario**. Aunque los modelos pueden generalizar hasta cierto punto, pueden fallar cuando se enfrentan a textos que contienen palabras o estructuras que no han aparecido con suficiente frecuencia en los datos de entrenamiento. Esto se observa en errores de clasificación cuando términos clave no están presentes o cuando el contexto semántico no se ha aprendido correctamente.

El modelo clásico tiene la limitación de **no captar relaciones complejas entre palabras**, ya que trabaja de manera más local sin considerar el contexto completo. Esto lo hace más propenso a errores cuando se trata de clasificar textos donde el significado depende de la relación entre múltiples palabras. En contraste, aunque el modelo basado en **Transformers** mejora este aspecto al utilizar mecanismos de atención, también tiene la desventaja de ser **computacionalmente más costoso**, lo que implica un mayor tiempo de entrenamiento y un mayor consumo de memoria en comparación con modelos más simples.

Por último, ninguno de los modelos ha sido entrenado con **un enfoque multilingüe**, por lo que su desempeño fuera del idioma en el que fue entrenado es incierto. Además, el modelo puede no reconocer correctamente términos técnicos, abreviaturas o jergas específicas que no estuvieron representadas en el conjunto de datos, lo que limita su aplicabilidad en contextos más especializados o dinámicos donde el lenguaje evoluciona rápidamente.

2.9 Pregunta 9

2.9.1 ¿Qué sería necesario para que este modelo pueda interpretar textos en español?

Para que este modelo pueda interpretar textos en español, sería necesario realizar varios ajustes en el proceso de entrenamiento y en la configuración del preprocesamiento de datos.

El primer paso sería utilizar un **conjunto de datos en español**, ya que el modelo ha sido entrenado con textos en inglés y su conocimiento está limitado a las estructuras lingüísticas y vocabulario de ese idioma. Entrenar el modelo con un corpus extenso en español le permitiría aprender la semántica, la gramática y las relaciones contextuales propias del idioma.

Otro aspecto fundamental sería **adaptar el proceso de tokenización y vectorización**. El actual `vectorizer` está basado en palabras y estructuras del inglés, por lo que sería necesario utilizar un **modelo de tokenización entrenado en español**, como el de *spaCy* para español o un `TextVectorization` adaptado a un nuevo corpus. Esto aseguraría que el modelo procese correctamente las palabras, considerando características particulares del idioma como los acentos, los pronombres y las conjugaciones verbales.

En el caso del modelo basado en **Transformers**, podría beneficiarse del uso de embeddings preentrenados específicos para español, como los de **FastText**, **word2vec en español** o incluso modelos avanzados como **BETO**, que es una versión en español de BERT. Estos *embeddings* ya contienen representaciones vectoriales optimizadas para el idioma, lo que permitiría que el modelo aprenda con mayor rapidez y precisión.

Además, sería recomendable reajustar los hiperparámetros y la arquitectura para asegurar que el modelo capture correctamente las características sintácticas y semánticas del español. Esto incluye ajustar la dimensión de los embeddings, la estrategia de atención y los métodos de regularización para evitar sobreajuste.

Por último, si el modelo fuera a usarse en un contexto multilingüe, una opción viable sería **entrenarlo con un *dataset* bilingüe o multilingüe**, permitiendo que aprenda patrones de transferencia entre idiomas. Modelos de Transformers como **mBERT** o **XLM-R** ya han sido preentrenados en múltiples idiomas y podrían ser utilizados como base para la clasificación de textos en español sin necesidad de un entrenamiento desde cero.