



UNIVERSIDAD
POLITÉCNICA
DE MADRID

Práctica II: Usos avanzados de transformers (LoRA y RAG)

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería de Sistemas Informáticos
(ETSIISI)

Mora Medina, Manuel Adrian

Sánchez De La Rosa Fuentes, Mario

Asignatura: Métodos Generativos

Fecha de entrega: Domingo, 30 de noviembre de 2025

Índice

1	Introducción	4
2	Parte I: LoRA	5
2.1	Introducción	5
2.2	Descripción del problema	6
2.3	El Dataset que usaremos	7
2.3.1	Extracción de las reseñas	7
2.3.2	Pasarlas al formato que acepta hugging face	8
3	Búsqueda de modelos	9
3.1	¿Con cuál nos quedamos?	9
3.2	Análisis en profundidad de ambos modelos	11
3.2.1	Análisis de Deberta	11
3.3	Análisis de Roberta	13
4	Entrenamiento con LoRA	15
4.1	LoRA para Deberta	15
4.1.1	Primer intento: LoRA con r=8, alpha=32, aplicado a Q y V, 2 epochs	18
4.1.2	Segundo intento: LoRA con r=8, alpha=32, aplicado a Q, K y V, 3 epochs	20
4.1.3	Tercer intento: LoRA con r=4, alpha=16, aplicado a Q, K y V, 3 epochs	22
4.1.4	Cuarto intento: LoRA con r=4, alpha=32, aplicado a Q, K y V, 3 epochs	24
4.1.5	Quinto y último intento: LoRA con r=4, alpha=32, aplicado a Q y V, 3 epochs	26
4.2	LoRA para Roberta	28
4.2.1	Primer intento: LoRA con r=8, alpha=32, aplicado a Q y V, 3 epochs	28
4.2.2	Segundo intento: LoRA con r=8, alpha=32, aplicado a Q, K y V, 3 epochs	30
4.2.3	Tercer intento: LoRA con r=4, alpha=16, aplicado a Q, K y V, 3 epochs	31
4.2.4	Cuarto intento: LoRA con r=4, alpha=32, aplicado a Q, K y V, 3 epochs	33
4.2.5	Quinto y último intento: LoRA con r=4, alpha=32, aplicado a Q y V, 3 epochs	35
5	Comparación y justificación de resultados	38

6 Extra: Comparación con PROMPT-ENGINEERING	41
6.1 Búsqueda de modelos generativos	41
6.2 Zero-shot	41
6.3 Few-shot	43
6.4 Conclusiones de esta comparación	44
7 RAG: Retrieval and Augmented Generation	46
8 Elección del corpus de conocimiento	47
9 Preparación del entorno	48
9.1 Aclaraciones	48
9.2 Carga de los modelos generadores de embeddings	48
9.2.1 Carga del modelo MPNET	48
9.2.2 Carga del modelo Jina	49
9.2.3 Comparación de ambos modelos	49
9.3 Búsqueda de vectores similares	50
9.3.1 Similitud coseno	50
9.3.2 Similitud euclídea	51
9.4 Sobre nuestro documento	52
9.4.1 Limpiar el documento	52
9.4.2 Crear chunks	53
9.5 Tokenizar los chunks	54
10 Preparar el entorno de pruebas	55
11 Búsqueda de modelos generadores	59
12 Probando las parejas	60
12.1 MPNET y QWEN: La pareja normal	60
12.1.1 Probando con similitud coseno, con chunk de tamaño 300, remember de 100 y top_k=10	60
12.1.2 Probando con similitud euclídea, con chunk de tamaño 300, remember de 100 y top_k=10	61
12.1.3 Probando con similitud coseno, con chunk de tamaño 500, remember de 200 y top_k=20	61
12.1.4 Probando con similitud euclídea, con chunk de tamaño 500, remember de 200 y top_k=20	62
12.1.5 Conclusiones con la pregunta básica (modelos normales)	62
12.1.6 Probando ahora con la pregunta especializada (modelos normales) .	62
12.1.7 Conclusiones de la pareja 'normal'	63

12.2 JINA y PHI: Los modelos 'especiales'	64
12.2.1 Probando con similitud coseno, con chunk de tamaño 300, remember de 100 y top_k=10	64
12.2.2 Probando con similitud euclidea, con chunk de tamaño 300, remem- ber de 100 y top_k=10	65
12.2.3 Omitimos las pruebas con los chunks	65
12.2.4 Conclusiones con la pregunta normal (modelos especiales)	65
12.2.5 Probando ahora con la pregunta especializada (modelos especiales)	65
12.2.6 Conclusiones de la pareja 'especial'	66
12.3 Conclusiones generales	66
13 Generando respuestas con ambas parejas	67
13.1 Respuestas de la pareja 'normal'	67
13.2 Respuestas de la pareja 'especial'	68
13.3 Conclusiones finales de las respuestas	69
14 Conclusiones finales	70
15 Referencias	71

1. Introducción

En esta práctica, vamos a aprender como usar los últimos modelos generativos (transformers: encoder, decoder o encoder-decoder), como encontrar el mejor modelo para nuestro caso (gracias a *hugging face*), especializarlos para una tarea específica y entrenarlos de tal manera que no consuma recursos.

Los transformers son una gran ventaja en comparación a las redes neuronales recurrentes, ya que, gracias a las cabezas de las capas de atención, podemos paralelizar el entrenamiento de estos modelos (cosa que no se podía con las RNN) sin perder la información temporal (contexto). Lo malo es que estos, tienen demasiados parámetros (pesos), por lo cual, necesitaríamos de un hardware y recursos tan grandes que no valdría la pena usarlos. Por suerte, tenemos técnicas para evitar estas limitaciones y entrenar menos pesos, o llegando incluso a no entrenar nada, pero siendo los modelos capaces de obtener información reciente (no hay necesidad de entrenar de nuevo estos modelos). Veremos en esta memoria 2 casos de como aplicar dichas técnicas, una para clasificar reseñas de Steam de un juego, y otra para sin necesidad de entrenar de nuevo, separa responder preguntas (como un QA) sobre un tema que no se le entrenó. Exploraremos también algunas comparaciones con otras técnicas, y exploraremos alguna que otra cosa interesante.

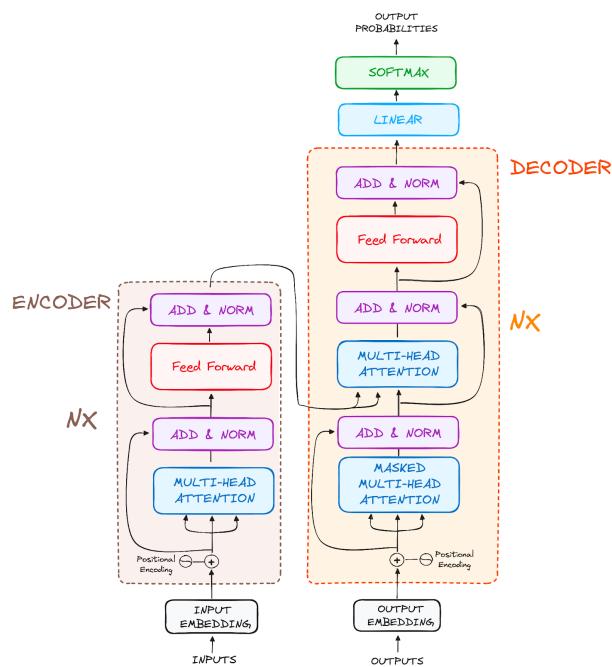


Figura 1: Arquitectura transformer.

2. Parte I: LoRA

2.1. Introducción

Hemos dicho que entrenar un modelo transformer no es viable para todo el mundo: Es necesario contar con grandes recursos que no están a la disposición de todos. Sabiendo esto, la técnica lora consiste en **entrenar solo una pequeña parte de los pesos (llegando a ser a veces menos de un 1%) para evitar el consumo de muchos recursos**. La gran ventaja de esto es que, podemos especializar un gran modelo solo entrenando una pequeña parte de este, y lo mejor, con resultados muy similares (visto en clase). LoRA lo que hace es formar 2 matrices de dimensiones m^*r y r^*n , partiendo que nuestra matriz de pesos original es m^*n . Esto nos crea 2 matrices más pequeñas que llegan a ser menos pesadas a la hora de entrenar. Lo bueno de esta técnica, nosotros podemos decidir a **qué vectores aplicarle**, ya que LoRA (las matrices) se aplican a los vectores q , k y v . Podemos jugar con esto y ver que resultados tenemos, y así entrenar y especializar con menos recursos un modelo capaz de hacer grandes cosas.

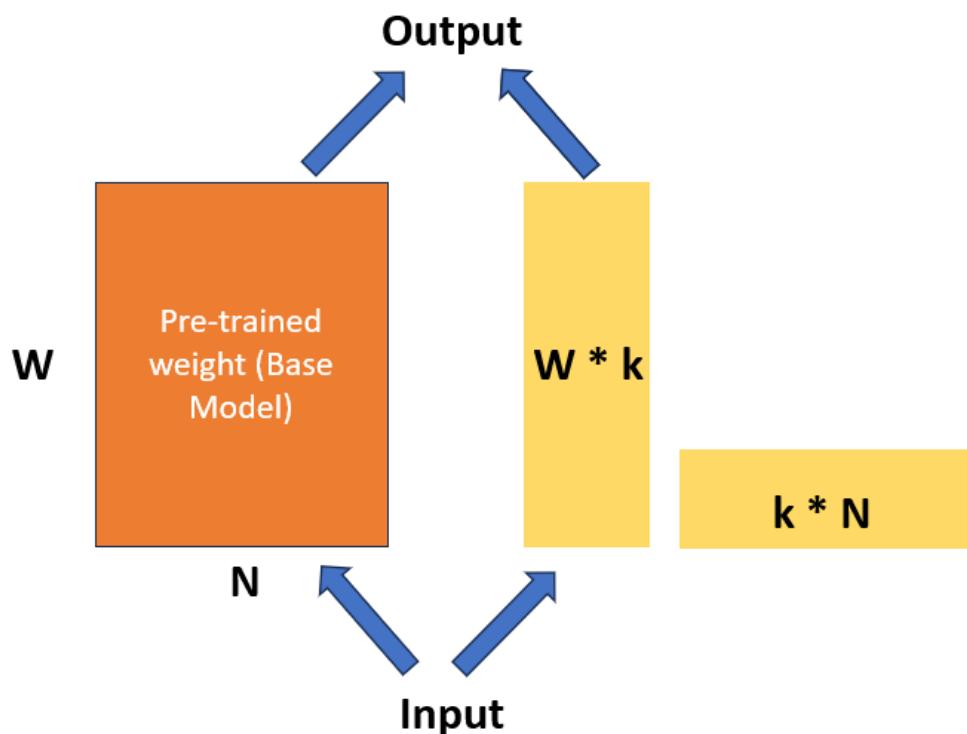


Figura 2: Ejemplo visual de LoRA.

2.2. Descripción del problema

Vamos a usar un modelo **encoder-only**, el cual, será capaz de clasificar reseñas de Steam de un videojuego llamado 'Geometry Dash'. Para ello, vamos a aclarar algunas cosas:

- ★ **¿Por qué un 'encoder-only'?** Porque es muy complicado entrenar un decoder-only con LoRA (al menos, cuando lo buscamos nos dió algo de miedo), y un encoder-decoder, lo pensamos pero nos gustó un poco más lo del encoder-only porque nos parece muy interesante todo el tema de análisis de sentimiento, y que mejor entrenando un modelo que se enfrente a algo tan complicado y confuso como reseñas (y si es de un videojuego, lo que te puedes encontrar).
- ★ **¿Por qué un dataset de reseñas de juegos de Steam?** Si, a lo mejor se nos critica por usar un dataset de un videojuego (no sabemos), pero lo que sí sabemos es que, en la vida real, los datos estarán muy 'sucios', es decir, tendrán cosas extrañas: Existirán comentarios con doble interpretación, muy coloquiales, faltas de ortografías, etc. Lo segundo: Steam solo da dos opciones para sus reseñas: positiva o negativa. Esto facilita (digamos) un poco la tarea del encoder, ya que tenemos **solo dos clases para poder clasificar las reseñas**. Lo malo es que, el modelo puede verse confundido con comentario extraños como ya vimos. Al no existir más clases como 'neutro', 'muy negativo', etc, pues es incluso más fácil poder equivocarse.
- ★ **¿Por qué reseñas de un videojuego?** La gente cuando va a comprar juegos (y luego algunos dar su opinión) suele expresarse muy mal en estos casos como ya vimos. Si vamos a otras secciones como reviews de algunos cursos y tal, al modelo no se le complicará tanto la tarea de clasificación al encontrar mejores reseñas y más entendibles. Por el contrario, con una reseña de un videojuego, no encontraremos este tipo de comentarios, por lo cual, le dará más tarea al modelo a la hora de clasificar.
- ★ **Longitud del dataset:** En total, hay casi **500k de reseñas, 400k positivas y 40k negativas**. Es suficiente para entrenar nuestro modelo, añadiendo que Python cuenta con una herramienta que nos ayudará a la hora de filtrar solo reseñas en inglés, separando las positivas de las negativas.

Aclarado esto, usaremos este dataset para entrenar un modelo **Encoder-only** para '**Análisis de sentimiento**'. Aclarado todo esto, vamos a ver los pasos que seguiremos para obtener el modelo de clasificación de reseñas.



Figura 3: Total de reseñas del juego Geometry Dash.

2.3. El Dataset que usaremos

Ya vimos que usaremos reseñas del juego 'Geometry Dash' de Steam. El objetivo ahora es extraer todas las reseñas posibles, y hemos decidido pasarlas **tal cual**, a ver qué resultados nos puede sacar (es posible que para algunos, cosas como los emojis sean algo bueno o malo, vamos a ver qué hace el modelo).



Figura 4: Ejemplo de reseñas de Geometry Dash en Steam.

2.3.1. Extracción de las reseñas

Para ello, usaremos una librería llamada 'steam_reviews.ReviewLoader', la cual descargará las reseñas según le especifiquemos idioma, tipo, etc.

```
# Función que devuelve un objeto con reseñas del juego
def obtener_tipo_reseñas(id_juego, tipo_reseña, num_reseñas=10000):
    # Instanciamos el objeto que nos descargara las reseñas
    loader = ReviewLoader()

    # Lo configuramos
    loader.set_language('english') # Idioma de las reseñas
    loader.set_review_type(tipo_reseña) # Tipo de reseñas: positivas o negativas
    loader.set_num_per_page(100) # Cuentas reseñas por página

    # Realizamos la descarga de las reseñas
    reviews = loader.load_batch_from_api(id_juego, num=num_reseñas)[0].review_dict()

    # Devolvemos las reseñas
    return reviews

```

0s

```
# Configuramos los datos para extraer la información de las reseñas
id_geometry = 322178

# Obtenemos las reseñas
positive_reviews = obtener_tipo_reseñas(id_geometry, 'positive')
negative_reviews = obtener_tipo_reseñas(id_geometry, 'negative')

```

1m24.4s

```
[0s] | 0/10000 [00:00< 7it/s]
Start the request. The game has total 300000 reviews, and will only get 10000 reviews.
10000 [███████] | 10000/10000 [00:34<0:00, 288.07it/s]
[0s] | 0/10000 [00:00< 7it/s]
Start the request. The game has total 300000 reviews, and will only get 10000 reviews.
10000 [███████] | 10000/10000 [00:35<0:00, 297.66it/s]
Requests count is 100, will wait for 15 seconds.
100it [00:48, 297.14it/s]
```

(a) Código para obtener las reseñas.

```
# Extraemos una reseña positiva y otra negativa
first_positive_review = positive_reviews[0]
first_negative_review = negative_reviews[0]

# Mostramos número de reseñas
print('*' * 15, 'Información reseñas', '*' * 15)
print('Total de reseñas: ', len(positive_reviews) + len(negative_reviews))
print('Total reseñas positivas: ', len(positive_reviews))
print('Total reseñas negativas: ', len(negative_reviews))

# Mostramos las info's básicas de las reseñas
print('*' * 15, 'Información reseña positiva', '*' * 15)
mostrar_info_reseña(first_positive_review)
print('*' * 15, 'Información reseña negativa', '*' * 15)
mostrar_info_reseña(first_negative_review)

=====
===== Información reseñas =====
Total de reseñas: 20000
Total reseñas positivas: 10000
Total reseñas negativas: 10000
===== Información reseña positiva =====
Idioma: english
¿Positiva?: True
Reseña:
cube jump spike fun
===== Información reseña negativa =====
Idioma: english
¿Positiva?: False
Reseña:
I hate this game
```

(b) Ejemplo de las reseñas.

Figura 5: Obtención y visualización de reseñas de Steam.

2.3.2. Pasarlas al formato que acepta hugging face

Esto es más fácil: solo es pasar las reseñas (dataframe) a un Dataset (para hugging face):

```
Con esto, ya podemos construir un dataframe:

# Creamos el diccionario final
reviews = {'reviews': list(), 'label': list()}

# Juntamos ambas reseñas y formamos un dataframe
for positive_review, negative_review in zip(positive_reviews, negative_reviews):

    # Añadimos la review positiva
    reviews['reviews'].append(positive_review['review'])
    reviews['label'].append(1 if positive_review['voted_up'] else 0)

    # Ahora la negativa
    reviews['reviews'].append(negative_review['review'])
    reviews['label'].append(1 if negative_review['voted_up'] else 0)

# Pasamos esto a un dataframe
reviews_dataframe = pd.DataFrame(reviews)
reviews_dataframe
```

0s

(a) Código para pasar reseñas a DataFrame.

```
Pero para los modelos de hugging face, hay que pasarlo a un tipo especial: Dataset.

reviews_hugging_face = Dataset.from_pandas(reviews_dataframe)
reviews_hugging_face
```

0s

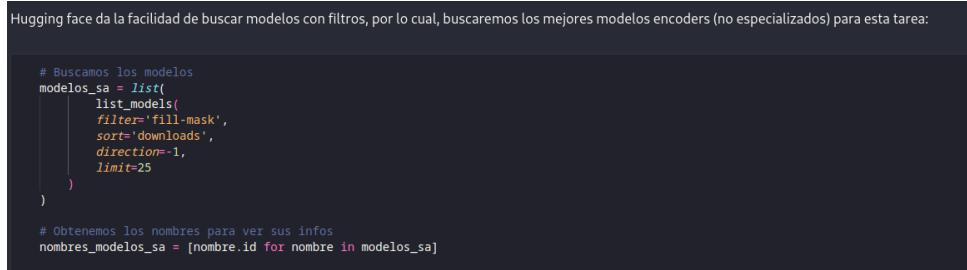
```
Dataset({
    features: ['reviews', 'label'],
    num_rows: 20000
})
```

(b) Código para pasar reseñas a Dataset (HF).

Figura 6: Reseñas en distintos formatos.

3. Búsqueda de modelos

Esta parte es importante, porque necesitamos encontrar los modelos que **no estén especializados en análisis de sentimiento**. También que tengán una 'Base de datos vectorial' sobre palabras **en inglés**, es decir, que sepa el vocabulario inglés. Para ello, debemos de configurar de manera adecuada la función 'list_models' de hugging face para obtener lo que necesitamos:



```
# Buscamos los modelos
modelos_sa = list(
    list_models(
        filter='fill-mask',
        sort='downloads',
        direction=-1,
        limit=25
    )
)
# Obtenemos los nombres para ver sus infos
nombres_modelos_sa = [nombre.id for nombre in modelos_sa]
```

Figura 7: List model para encontrar el encoder adecuado.

Esto es lo que nos salió luego de la búsqueda:

	nombre	autor	tarea	fecha-creacion	fecha-ultima-actualizacion	descargas	likes	tags	licencia
0	FacebookAI/roberta-large	FacebookAI	fill-mask	2022-03-02 23:29:04+00:00	2024-02-19 12:47:04+00:00	15547966	256	transformers, pytorch, tf, jax, onnx, safetensors...	mit
1	FacebookAI/roberta-base	FacebookAI	fill-mask	2022-03-02 23:29:04+00:00	2024-02-19 12:39:28+00:00	9111588	534	transformers, pytorch, tf, jax, rust, safetensors...	mit
2	FacebookAI/xlm-roberta-base	FacebookAI	fill-mask	2022-03-02 23:29:04+00:00	2024-02-19 11:48:21+00:00	7454216	762	transformers, pytorch, tf, jax, onnx, safetensors...	mit
3	google-bert/bert-base-multilingual-cased	google-bert	fill-mask	2022-03-02 23:29:04+00:00	2024-02-19 11:05:41+00:00	5904268	554	transformers, pytorch, tf, jax, safetensors, b...	apache-2.0
4	google-bert/bert-base-multilingual-uncased	google-bert	fill-mask	2022-03-02 23:29:04+00:00	2022-04-06 11:06:00+00:00	4945252	142	transformers, pytorch, tf, jax, safetensors, b...	apache-2.0
5	nlpaeub/legal-bert-base-uncased	nlpaeub	fill-mask	2022-03-02 23:29:05+00:00	14:42:50+00:00	4297797	287	transformers, pytorch, tf, jax, bert, pretrain...	cc-by-sa-4.0
6	google-bert/bert-base-cased	google-bert	fill-mask	2022-03-02 23:29:04+00:00	2024-02-19 11:02:26+00:00	3580155	336	transformers, pytorch, tf, jax, safetensors, b...	apache-2.0
7	FacebookAI/xlm-roberta-large	FacebookAI	fill-mask	2022-03-02 23:29:04+00:00	2024-02-19 12:48:30+00:00	3128981	474	transformers, pytorch, tf, jax, onnx, safetens...	mit
8	emilyalsentzer/Bio_ClinicalBERT	emilyalsentzer	fill-mask	2022-03-02 23:29:05+00:00	2024-12-03 20:22:45+00:00	2938591	400	transformers, pytorch, tf, jax, bert, fill-mas...	mit
9	facebook/esm2_133_650M_URSD0	facebook	fill-mask	2022-09-27 14:36:16+00:00	2023-03-21 15:05:12+00:00	2913957	59	transformers, pytorch, tf, safetensors, esm, f...	mit
10	microsoft/deberta-v3-base	microsoft	fill-mask	2022-03-02 23:29:05+00:00	2024-02-19 12:34:19+00:00	1999363	376	transformers, pytorch, tf, rust, deberta-v2, d...	mit
11	google-bert/bert-base-chinese	google-bert	fill-mask	2022-03-02 23:29:04+00:00	2025-07-03 11:58:48+00:00	1693973	1334	transformers, pytorch, tf, jax, safetensors, b...	apache-2.0
12	dandelin/vilt-b32-mm	dandelin	fill-mask	2022-03-02 23:29:05+00:00	2022-07-06 12:18:37+00:00	1582757	12	transformers, pytorch, vilt, fill-mask, arxiv...	apache-2.0
13	distilbert/distilroberta-base	distilbert	fill-mask	2022-03-02 23:29:04+00:00	2024-02-19 11:09:58+00:00	1578502	164	transformers, pytorch, tf, jax, rust, safetens...	apache-2.0
14	facebook/esm2_t6_8M_URSD0	facebook	fill-mask	2022-09-26 18:44:55+00:00	2023-03-21 15:05:17+00:00	1431301	24	transformers, pytorch, tf, safetensors, esm, f...	mit
15	microsoft/BioMedNLP-BioMedBERT-base-uncased-ab...	microsoft	fill-mask	2022-03-02 23:29:05+00:00	2023-11-06 18:04:15+00:00	1179694	83	transformers, pytorch, jax, bert, fill-mask, e...	mit
16	InstaDeepAI/nucleotide-transformer-500m-human...	InstaDeepAI	fill-mask	2023-04-04 21:37:57+00:00	2024-07-22 09:23:44+00:00	99846	14	transformers, pytorch, tf, joblib, esm, fill-m...	cc-by-nc-sa-4.0
17	dccuchile/bert-base-spanish-wmvm-uncased	dccuchile	fill-mask	2022-03-02 23:29:05+00:00	2024-01-18 01:46:43+00:00	984552	72	transformers, pytorch, tf, jax, bert, fill-mas...	None

Figura 8: Algunos resultados de la búsqueda de modelos.

NOTA: Hemos aplicado filtros como evitar usar modelos ya vistos en clase, que solo sean del tipo 'fill-mask' (sin especializar), y algunas cosas más. Ver el notebook que ahí se mira los filtros.

3.1. ¿Con cuál nos quedamos?

Hemos estado viendo de los que nos salió cuales pueden funcionar mejor, y tras investigar (y según ChatGPT), los modelos **FacebookAI/roberta-base** y **microsoft/deberta-v3-base** son los más indicados para la tarea. Otro motivo es que el primero está algo más

actualizado (su última actualización fue en 2024) que el segundo (2022). Seleccionados estos modelos, vamos a analizarlos más en profundidad.

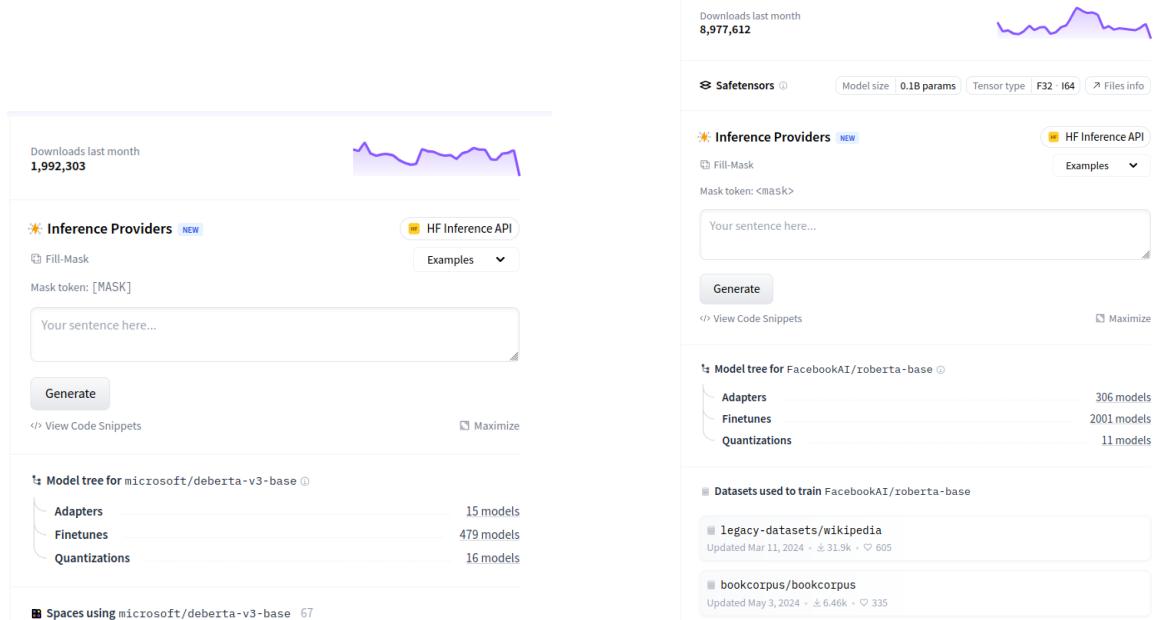


Figura 9: Ambos modelos (Deberta y Roberta) en HF.

3.2. Análisis en profundidad de ambos modelos

Vamos a analizar en profundidad cada modelo para conocer la arquitectura base, si tiene licencia, pesos y tamaño del modelo, técnicas de preentrenamiento utilizadas. Veremos otra información como el tipo de datos usado para entrenar, autores, fechas de creación y actualización, entre otras más.

	nombre	autor	tarea	fecha-creacion	fecha-ultima-actualizacion	arquitectura_base	tipo_modelo	parametros	peso_disco	licencia	tecnicas_preentrenamiento	descargas	tarea_principal	likes
0	FacebookAI/roberta-base	FacebookAI	fill-mask	2022-03-02 23:29:04+00:00	2024-02-19 12:39:28+00:00	RobertaForMaskedLM	roberta	124647170	2684.78 MB (2.62 GB)	mit	Ver Model Card (Texto)	9111588	fill-mask	534
1	microsoft/deberta-v3-base	microsoft	fill-mask	2022-03-02 23:29:05+00:00	2022-09-22 12:34:19+00:00	Desconocida	deberta-v2	184423682	1765.66 MB (1.72 GB)	mit	Ver Model Card (Texto)	1999363	fill-mask	376

Información básica:

- Nº de parámetros: Una diferencia normalita (~185M de parámetros contra ~125M de parámetros).
- Arquitectura base: El más pesado (deberta) tiene una arquitectura desconocida (no se pudo encontrar), pero según Copilot, está basada en RoBERTa (basada en BERT) y roberta, en RobertaForMaskedLM.
- Licencia: Ambos tienen la licencia mit (software libre y de código abierto).
- Peso del modelo: 1.72 contra 2.62GB. Curioso porque deberta tiene más parámetros que roberta.
- Técnicas de pre-entrenamiento: Ambos se pre-entrenaron con la técnica de Ver Model Card (Texto).

Información extra recomendada:

- Autor: microsoft y FacebookAI (Dos grandes empresas).
- Fecha de creación: Aunque ambos son del mismo año y mes (2022-03-02), roberta está más actualizado (2024 vs 2022).
- Descargas: Roberta ha sido probado por mucha más gente (~10M descargas vs ~2M descargas de Deberta).
- Ambos no tienen cabezas especializadas (para sentiment-analyzer, clasificación, etc.).

Figura 10: Información de ambos modelos.

3.2.1. Análisis de Deberta

Vamos a profundizar en los datos de Deberta:

- **Número de parámetros:** Vamos por lo más importante: Los pesos del modelo. Vimos que tiene casi 185M de pesos. Esto se debe porque deberta tiene **12 capas encoder, cada una con 12 cabezas**. Las cabezas son capas de atención, solo que tratan parte de los vectores. Es decir, como sabemos, las palabras sufren un proceso de **tokenización**, donde las palabras pasan a ser IDs, y esas IDs, cada modelo tiene su propia 'base de datos vectorial'. En este caso, deberta tiene una BBDD vectorial grande, de 768. O sea, que cada palabra es un 'tensor' de dimensión 768. Con todo eso, es normal que deberta tenga tantos parámetros/pesos, al existir muchas capas encoder y cabezas en cada una de estas. Y no hemos tomado en cuenta la capa de feed forward, la cual son **MLPs con sus pesos**. Con todo esto, es normal que tengamos muchos pesos.
- **Arquitectura base:** Ya lo vimos, dice desconocida, pero buscando en internet (y según su nombre), esta basada en BERT (casi todos o la mayoría de encoders están basados en BERT pero fine-tuneados). No hay más que decir, ya que este es un modelo fine-tunning de BERT.
- **Licencia:** La licencia que tiene se le llama mit, la cual es de **software libre**, es decir, que podemos ver su código, modificarlo, explorarlo, etc. Esto

es bueno, ya que podemos ver como se ha programado y diseñado este modelo transformer.

- **Peso/tamaño del modelo:** Vimos que es tocho el modelo, y no es de sorprender cuanto nos ocuparía en RAM... ¡Más de 1.5GB (1.72 aproximadamente)! Normal con tantos pesos y capas MLP que nos pese mucho. Aún así, no es lo más pesado en general (BERT pesa mucho más).
- **Técnicas de pre-entrenamiento:** Según HF, Deberta ha sido entrenada con una técnica llamada **Masked language modeling (MLM)**:

Nota importante

DeBERTa improves the BERT and RoBERTa models using disentangled attention and enhanced mask decoder. With those two improvements, DeBERTa outperforms RoBERTa on a majority of NLU tasks with 80GB training data.

Esto quiere decir que (si inferimos el texto), el modelo trata de predecir el token según un contexto. Con esto, solo nos queda especializarlo para predecir el contexto (sentimiento).

- **Autor:** El autor de este modelo es Microsoft, un competidor algo lejano en cuanto empresas que apuestan por la IA, pero bueno.
- **Fecha de creación:** Fue creado el 02-03-2022, y su última actualización no pasa del 2022. Se puede decir que está desactualizado.
- **Datasets de entrenamiento:** Lamentablemente no se dice con qué datasets se entrenó (cosa contraria con Roberta), pero solo nos dice que se entrenó **con 80GB de datos**, una barbaridad. Pero si buscamos en sus papers (link en referencias):

Nota importante

As an important extension, we extend DeBERTaV3 to multi-lingual. We train the multi-lingual model with the 2.5T CC100 multi-lingual dataset which is the same as XLM-R. We denote the model as mDeBERTaV3base. We use the same SentencePiece vocabulary as mT5 which has 250k tokens.

O sea, que aparte de ser multilingüe, también nos dice que se entrenó con el dataset de **CC100 Multi-lingual**. Este dataset es gigante, con 2.5T de tokens, y lo interesante es que hace web scrapping (sacar información de las webs) para convertirla en texto. Un dataset muy fiable.

```
Cabezas de cada capa: 12
Formato de guardado de pesos: torch.float32
Estructura de roberta:
DebertaV2ForSequenceClassification(
(deberta): DebertaV2Model(
(embeddings): DebertaV2Embeddings(
(word_embeddings): Embedding(128100, 768, padding_idx=0)
(LayerNorm): LayerNorm((768,), eps=1e-07, elementwise_affine=True)
(dropout): Dropout(p=0.1, inplace=False)
)
(encoder): DebertaV2Encoder(
(layer): ModuleList(
(0-11): 12 x DebertaV2Layer(
(attention): DebertaV2Attention(
(self): DisentangledSelfAttention(
(query_proj): lora.Linear(
(base_layer): Linear(in_features=768, out_features=768, bias=True)
(lora_dropout): ModuleDict(
(default): Dropout(p=0.1, inplace=False)
)
(lora_A): ModuleDict(
(default): Linear(in_features=768, out_features=4, bias=False)
)
(lora_B): ModuleDict(
(default): Linear(in_features=4, out_features=768, bias=False)
...
)
(dropout): Dropout(p=0.1, inplace=False)
)
)
)
)
)
```

Figura 11: Información de Deberta algo más específica.

3.3. Análisis de Roberta

Para no repetir mucho, para cosas que sean iguales a Deberta, no nos explicaremos mucho:

- **Número de parámetros:** Tiene casi 125M de pesos. Muy parecido a Deberta, ya que tiene 12 capas encoder, de las cuales, tiene 12 cabezas también, y una BBDD vectorial de dimensión 768. La diferencia es que seguramente, tenga menos capas MLP, y seguramente de ahí sea la menoría de pesos.
- **Arquitectura base:** Nos lo decia la tabla, y su arquitectura base es **RobertFor-MaskedLM**. O sea, ya nos dice que se esta basada en un modelo pre-entrenado para MLM, y basado en Roberta (este es como el padre, el que usamos es el fine-tuneado).
- **Licencia:** La misma que Deberta.
- **Peso/tamaño del modelo:** Es curioso, porque con menos parámetros, ¡Pesa más que Deberta, 2.5GB casi! Esto puede ser porque alguno de seguramente guarda sus pesos en formatos diferentes (no supimos sacarlo).
- **Técnicas de pre-entrenamiento:** Según HF, Roberta ha sido entrenada con una técnica llamada **Masked language modeling (MLM)**:

Nota importante

Pretrained model on English language using a masked language modeling (MLM) objective. It was introduced in this paper and first released in this repository. This model is case-sensitive: it makes a difference between english and English.

- **Autor:** El autor de este modelo es Facebook, que bueno, también aportar al mercado de la IA (LLAMA, por ejemplo).
- **Fecha de creación:** Fue creado el 02-03-2022, y su última actualización esta por el 2024. Se puede decir que esta más actualizado que Deberta.

```
Cabezas de cada capa: 12
Formato de guardado de pesos: torch.float32
Estructura de roberta:
  RobertaForSequenceClassification(
    (roberta): RobertaModel(
      (embeddings): RobertaEmbeddings(
        (word_embeddings): Embedding(50265, 768, padding_idx=1)
        (position_embeddings): Embedding(514, 768, padding_idx=1)
        (token_type_embeddings): Embedding(1, 768)
        (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (encoder): RobertaEncoder(
        (layer): ModuleList(
          (0-11): 12 x RobertaLayer(
            (attention): RobertaAttention(
              (self): RobertaSdpSelfAttention(
                (query): lora.Linear(
                  (base_layer): Linear(in_features=768, out_features=768, bias=True)
                  (lora_dropout): ModuleDict(
                    (default): Dropout(p=0.1, inplace=False)
                  )
                  (lora_A): ModuleDict(
                    (default): Linear(in_features=768, out_features=4, bias=False)
                  )
                ...
              )
            )
          )
        )
      )
    )
  )
)
```

Figura 12: Información de Roberta algo más específica.

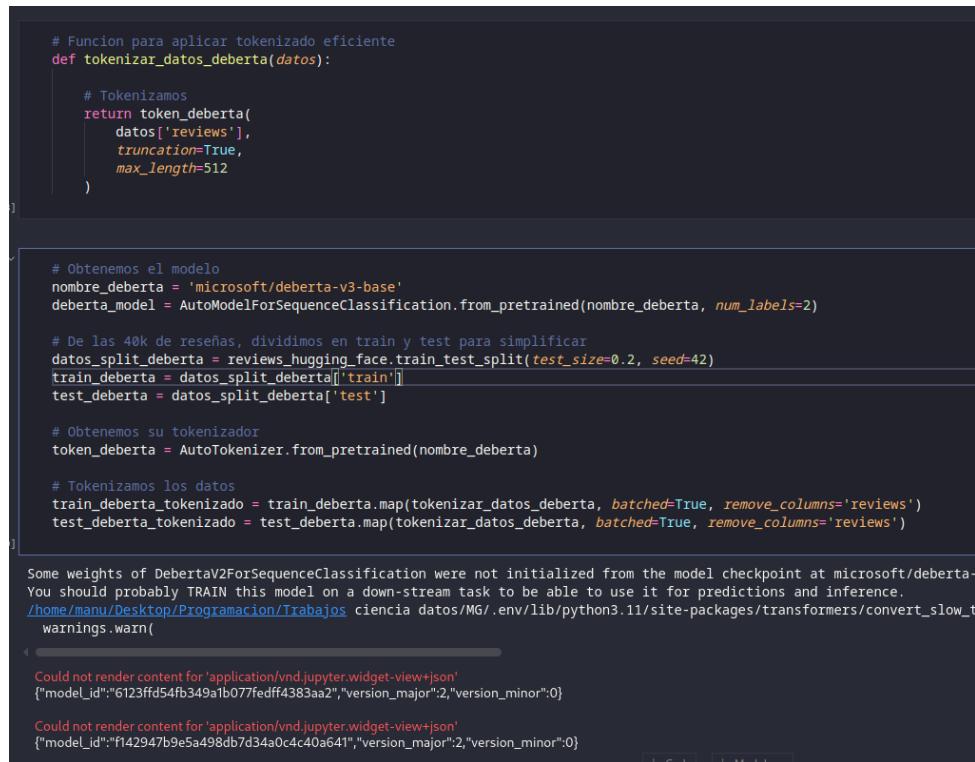
4. Entrenamiento con LoRA

Veremos ahora como vamos a especializar estos modelos para análisis de sentimiento de reseñas:

4.1. LoRA para Deberta

Para ello, debemos de seguir los siguientes pasos:

- **Tokenización de la entrada:** Debemos de pasar nuestras reseñas al lenguaje entendido por Deberta: Un vector de dimensión 768. Recordemos que dividimos en train y test:



```

# Funcion para aplicar tokenizado eficiente
def tokenizar_datos_deberta(datos):

    # Tokenizamos
    return token_deberta(
        datos['reviews'],
        truncation=True,
        max_length=512
    )

# Obtenemos el modelo
nombre_deberta = 'microsoft/deberta-v3-base'
deberta_model = AutoModelForSequenceClassification.from_pretrained(nombre_deberta, num_labels=2)

# De las 40k de reseñas, dividimos en train y test para simplificar
datos_split_deberta = reviews_hugging_face.train_test_split(test_size=0.2, seed=42)
train_deberta = datos_split_deberta['train']
test_deberta = datos_split_deberta['test']

# Obtenemos su tokenizador
token_deberta = AutoTokenizer.from_pretrained(nombre_deberta)

# Tokenizamos los datos
train_deberta_tokenizado = train_deberta.map(tokenizar_datos_deberta, batched=True, remove_columns='reviews')
test_deberta_tokenizado = test_deberta.map(tokenizar_datos_deberta, batched=True, remove_columns='reviews')

```

Some weights of DebertaV2ForSequenceClassification were not initialized from the model checkpoint at microsoft/deberta-v3-base. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

/home/manu/Desktop/Programación/Trabajos ciencia datos/MG/.env/lib/python3.11/site-packages/transformers/convert_slow_t
warnings.warn(

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id": "6123ffd54fb349a1b077fedff4383aa2", "version_major": 2, "version_minor": 0}

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id": "f142947b9e5a498db7d34a0c4c40a641", "version_major": 2, "version_minor": 0}

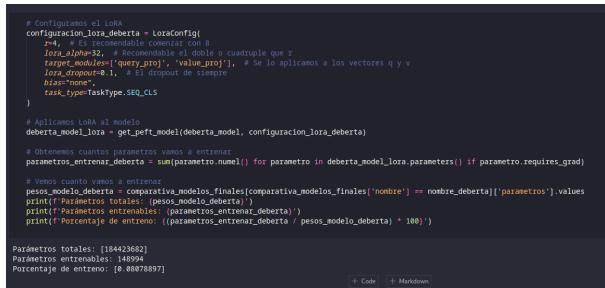
Figura 13: Resultado de la tokenización con Deberta (su tokenizador). Mirar que AutoModelForSequentialClassification hace la tarea de crear 2 cabezas para la predicción.

- **Configuración de las matrices (LoRA):** Para ello, vamos a usar la librería PEFT y transformer, que nos permiten configurar esto de manera más sencilla. A partir de aquí, se ha realizado unas cuantas pruebas de parámetros para ver los mejores resultados (en la siguiente sección se verá más detallado). De momento, recordemos esto:

- (a) **r:** Es como una dimensión de las matrices. Recordemos que en principio, tenemos una matriz 'w' de 'nxm' pesos. Para no romper dicha compatibilidad,

Creamos 2 matrices 'nxr' y 'rxm', así su producto nos da una matriz nxm. r suele ser un valor mayormente 8, pero podemos variarlo dependiendo de su rendimiento, pero si o si **debe de ser menor a n y m, sino no tendrá gracia (dos matrices de gran magnitud)**.

- (b) **lora_alpha:** Es la atención (importancia) que se le dará a la matriz a la hora de entrenar. En otras palabras, es como decirle al modelo 'Cuando entrenes, dale más importancia a lo nuevo que vas a aprender'. Este valor alpha suele ser 2^*r o 4^*r , pero la cosa es probar a ver que le puede dar mejor.
- (c) **dropout:** Son las capas de dropout que tendrá el modelo. Ya lo conocemos de antes.
- (d) **task_type:** La tarea que hará el modelo, en nuestro caso, clasificación con tokens secuenciales.
- (e) **target_modules:** Una cosa importantísima: A qué vectores se les aplicará la técnica (matrices). Esta matriz nueva se le puede aplicar a Q, K y V individualmente, ya es decisión nuestra a cuales le aplicamos (y cual rinde mejor). Es verdad que a k no se le suele aplicar, pero probaremos que tal le va.



```

# Configuramos el LoRA
configuracion_lora_deberta = LoraConfig(
    r=8, # Es recomendable tener entre 8 y 16
    lora_alpha=32, # Recomendable el doble o cuádruple que r
    target_modules=['query_proj', 'value_proj'], # Se lo aplicamos a los vectores q y v
    layerwise_output=True, # El dropout de siempre
    bias='none',
    task_type=TaskType.SEQ_CLS
)

# Aplicamos LoRA al modelo
deberta_model_lora = get_peft_model(deberta_model, configuracion_lora_deberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_deberta = sum([parametro.numel() for parametro in deberta_model_lora.parameters() if parametro.requires_grad])

# Vemos cuantos vamos a entrenar
pesos_comparativa = comparativa_modelos_finales['comparativa_modelos_finales']['nombre'] == nombre_deberta]['parametros'].values
print(f'Parametros totales: {pesos_comparativa["totales"]}, pesos trainable: {pesos_comparativa["trainable"]}')
print(f'Parametros entrenables: {parametros_entrenar_deberta}')
print(f'Porcentaje de entreno: {(parametros_entrenar_deberta / pesos_comparativa["totales"]) * 100} %')

Parámetros totales: [184423682]
Parámetros entrenables: 148994
Porcentaje de entreno: [0.00878897]

```

Figura 14: Ejemplo de como se vería una configuración de LoRA para Deberta.

- **Entrenamiento del modelo:** Una vez pasada la información de la matriz, ya podemos entrenar el transformer. Aunque se entrena muy pocos pesos, el entreno demorará un cierto tiempo debido a la cantidad masiva de pesos (en ambos casos más de 120M de parámetros).

```

# Métrica de evaluación
accuracy_metric = evaluate.load("accuracy")

# Esta función es del notebook de fine tuning de lora, como nos sirve, hemos decidido reusarla. No hata plis :(
def compute_metrics(eval_pred):

    # Sacamos predicciones y reales
    predictions, labels = eval_pred

    # Convertir logits a clases predichas (argmax sobre dimensión de clases)
    predictions = np.argmax(predictions, axis=1)

    # Calcular accuracy comparando predicciones con labels verdaderos
    return accuracy_metric.compute(predictions=predictions, references=labels)
}

# Preparamos los hiperparámetros del modelo
args_deberta = TrainingArguments(
    num_train_epochs=3, # Con 3 ya se muestra mucha potencia, con 4 o más puede llegar a overfitting
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    learning_rate=2e-4,
    weight_decay=0.01,
    eval_strategy='epoch',
    save_strategy='no',
    logging_steps=50,
    report_to='none',
    fp16=True
)

# Creamos el trainer (como lo que vamos a entrenar)
deberta_train = Trainer(
    model=deberta_model_lora,
    args=args_deberta,
    train_dataset=train_deberta_tokenizado,
    eval_dataset=test_deberta_tokenizado,
    processing_class=token_deberta,
    data_collator=data_collator_deberta,
    compute_metrics=compute_metrics
)

```

Figura 15: Ejemplo de como se entrenaria un modelo Deberta con LoRA.

- **Evaluación del modelo:** Finalmente, vamos a evaluar el modelo, viendo no solo métricas como el accuracy, sino más como la precisión (cuantas clases 1 es capaz de predecir correctamente), recall (cuantas clases 1 es capaz de identificar correctamente) y f1-score (basado en ponderar la precisión y recall). No solo lo haremos con el conjunto de test que guardamos, sino con reseñas personalizadas, a ver que tal le va.

```

# Predecimos las reseñas de test
pred_proba_deberta = deberta_train.predict(test_deberta_tokenizado) # Ya usamos esto como validacion, solo sacaremos mas metricas

# Lo convertimos a 1 o 0
pred_deberta = np.argmax(pred_proba_deberta.predictions, axis=1)

# Mostramos metricas mas avanzadas
metricas_deberta_eval = classification_report(y_true=test_deberta['label'], y_pred=pred_deberta)
print(metricas_deberta_eval)

```

	precision	recall	f1-score	support
0	0.84	0.87	0.85	2028
1	0.86	0.83	0.84	1972
accuracy			0.85	4000
macro avg	0.85	0.85	0.85	4000
weighted avg	0.85	0.85	0.85	4000


```

# Tokenizamos las reseñas
reseñas_eval_token_deberta = reseñas_evaluar_hugging.map(tokenizar_datos_deberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_deberta = deberta_train.predict(reseñas_eval_token_deberta)

# Vemos los resultados
pred_eval_deberta = np.argmax(pred_proba_eval_deberta.predictions, axis=1)
metricas_deberta_eval_personalizado = classification_report(y_true=pred_proba_eval_deberta.label_ids, y_pred=pred_eval_deberta)
print(metricas_deberta_eval_personalizado)

```

Map: 0% | 0/4 [00:00<?, ? examples/s]

	precision	recall	f1-score	support
0	1.00	1.00	1.00	2
1	1.00	1.00	1.00	2
accuracy			1.00	4
macro avg	1.00	1.00	1.00	4
weighted avg	1.00	1.00	1.00	4

Figura 16: Ejemplo de como se entrenaría un modelo Deberta con LoRA.

Dicho esto, veremos las pruebas realizadas para entrenar a Deberta (mirar el tiempo de ejecución de las celdas).

4.1.1. Primer intento: LoRA con r=8, alpha=32, aplicado a Q y V, 2 epochs

Para este primer intento, probamos una configuración básica.

```

    # Configuramos el LoRA
configuracion_lora_deberta = LoraConfig(
    r=8, # Es recomendable comenzar con 8
    lora_alpha=32, # Recomendable el doble o cuádruple que r
    target_modules=['query_proj', 'value_proj'], # Se lo aplicamos a los vectores q y v
    lora_dropout=0.1, # El dropout de siempre
    bias='none',
    task_type=TaskType.SEQ_CLS
)

# Aplicamos LoRA al modelo
deberta_model_lora = get_peft_model(deberta_model, configuracion_lora_deberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_deberta = sum(parametro.numel() for parametro in deberta_model_lora.parameters() if parametro.requires_grad)

# Vemos cuantos pesos tenemos
pesos_modelo_deberta = comparativa_modelos_finales['pesos_modelo_deberta']
print(f'Parámetros totales: {pesos_modelo_deberta}')
print(f'Parámetros entrenables: {parametros_entrenar_deberta}')
print(f'Porcentaje de entrenamiento: {(parametros_entrenar_deberta / pesos_modelo_deberta) * 100}')
```

[30] ✓ 0.0s

.. Parámetros totales: [184423682]
.. Parámetros entrenables: 296450
.. Porcentaje de entrenamiento: [0.160744]

+ Code + Markdown

Figura 17: Configuración LoRA Deberta inicial.

Como podemos ver, aplicando LoRA solo a los vectores q y v, tenemos que entrenar un 0.17% de los pesos (300k aproximadamente). Es una barbaridad menor que entrenar todos los pesos.

```

# Entrenamos
deberta_train.train()

[1] ✓ 6m19.5s
The tokenizer has new PAD/BOS/EOS tokens that differ from the model config and generation config. The model config and generation config were aligned accordingly, being updated.
[1] ✓ 6m19.5s [4000/4000 06:18, Epoch 2/2]
Epoch TrainingLoss ValidationLoss Accuracy
1 0.343700 0.390068 0.848250
2 0.357000 0.376133 0.854750

TrainOutput(global_step=4000, training_loss=4025615017414093, metrics={'train_runtime': 379.2779, 'train_samples_per_second': 84.371, 'train_steps_per_second': 10.546, 'total'})

+ Code + Markdown
```

Figura 18: Entrenamiento Deberta con la configuración inicial.

Cuando la ponemos a entrenar, con solo 2 epochs, vemos que tiene un accuracy de 0.85. Esta bastante bien, ya que en 2 epochs nuestro modelo prácticamente está ya especializado en el tema. Recordar que 2 epochs normalmente ya suele especializarse, con 3, afina detalles, pero con 4 o más (e incluso a veces 3), suele tender al overfitting. Para asegurarnos que no pase eso, veamos más métricas.

```

Ahora, vamos a ver algunas métricas con classification_report:

# Predecimos las reseñas de test
pred_proba_deberta = deberta_train.predict(test_deberta_tokenizado) # Ya usamos esto como validación, solo sacaremos más métricas

# Lo convertimos a 1 o 0
pred_deberta = np.argmax(pred_proba_deberta.predictions, axis=1)

# Mostaremos métricas más avanzadas
metricas_deberta_eval = classification_report(y_true=test_deberta['label'], y_pred=pred_deberta)
print(metricas_deberta_eval)
```

[4] ✓ 10.3s

	precision	recall	f1-score	support
0	0.86	0.83	0.84	2028
1	0.83	0.86	0.85	1972
accuracy			0.84	4000
macro avg	0.85	0.84	0.84	4000
weighted avg	0.85	0.84	0.84	4000

Figura 19: Evaluación LoRA Deberta Inicial.

Vemos que, en general, sabe predecir muy bien ambas clases, así que podemos decir

un 90% que nuestro modelo está especializado. Pero veamos una última parte, donde ponemos nuestras propias reseñas:

```
> ~
> # Tokenizamos las reseñas
> reseñas_eval_deberta = reseñas_evaluar_hugging.map(tokenizar_datos_deberta, batched=True, remove_columns='reviews')

> # Se lo pasamos al modelo
> pred_proba_eval_deberta = deberta_train.predict(reseñas_eval_deberta)

> # Vemos los resultados
> pred_eval_deberta = np.argmax(pred_proba_eval_deberta.predictions, axis=1)
> métricas_deberta_eval_personalizado = classification_report(y_true=pred_proba_eval_deberta.label_ids, y_pred=pred_eval_deberta)
> print(métricas_deberta_eval_personalizado)
127] 0.0s
...
Map: 100%|██████████| 4/4 [00:00<00:00, 1186.51 examples/s]
precision    recall   f1-score   support
          0       1.00     1.00     1.00      2
          1       1.00     1.00     1.00      2

accuracy                           1.00      4
macro avg       1.00     1.00     1.00      4
weighted avg    1.00     1.00     1.00      4
```

Figura 20: Evaluación personalizada LoRA Deberta Inicial.

Vemos que lo hace perfecto, por lo cual, aseguramos que ya nuestro modelo está especializado y es capaz de clasificar las reseñas. No obstante, veremos si con una matriz más pequeña (menor r), es posible conseguir similares o mejores resultados (para la gente con menos recursos).

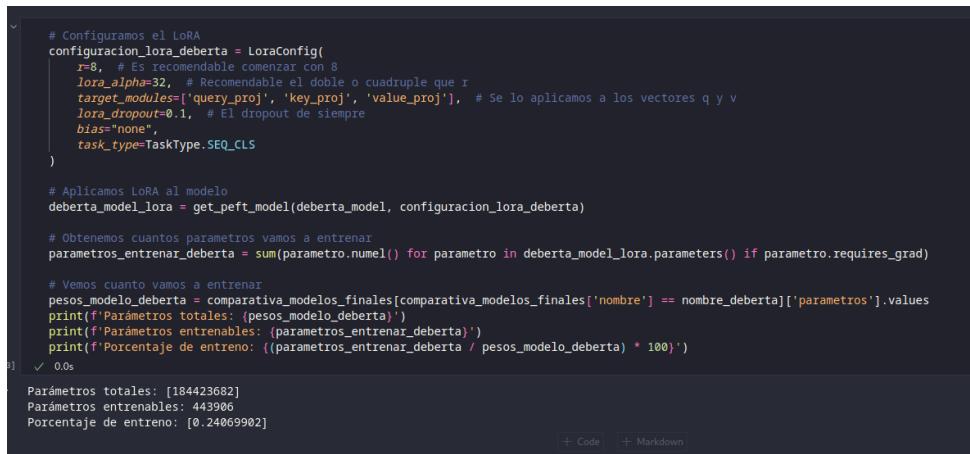
```
# Reseñas inventadas
reseñas_evaluar = {
    'reviews': [
        'This game is amazing, although the first levels are difficult, but then online levels are incredible, specially gauntlet.',
        'This poor shit is stressfull. I can't beat Stereo Madness because the fking cube jumped when I clicked. Delete this game please.',
        'Robot took 5 years to give us 2.2, but personally, I think the update is good, but for 5 years, is quite a long time.',
        'This game is amazing, but I am noob :( (phobos is hard)'
    ],
    'label': [1, 0, 0, 1]
}

# Pasamos esto a un dataset de hugging face
reseñas_evaluar_hugging = Dataset.from_dict(reseñas_evaluar)
```

Figura 21: Comentarios usados para la evaluación personalizada.

4.1.2. Segundo intento: LoRA con r=8, alpha=32, aplicado a Q, K y V, 3 epochs

Por curiosidad, quisimos ver que pasa si le aplicamos esto también al vector K (y añadiendo 3 epochs).



```

# Configuramos el LoRA
configuracion_lora_deberta = LoraConfig(
    r=8, # Es recomendable comenzar con 8
    lora_alpha=32, # Recomendable el doble o cuádruple que r
    target_modules=['query_proj', 'key_proj', 'value_proj'], # Se lo aplicamos a los vectores q y v
    lora_dropout=0.1, # El dropout de siempre
    bias='none',
    task_type=TaskType.SEQ_CLS
)

# Aplicamos LoRA al modelo
deberta_model_lora = get_peft_model(deberta_model, configuracion_lora_deberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_deberta = sum(parametro.numel() for parametro in deberta_model_lora.parameters() if parametro.requires_grad)

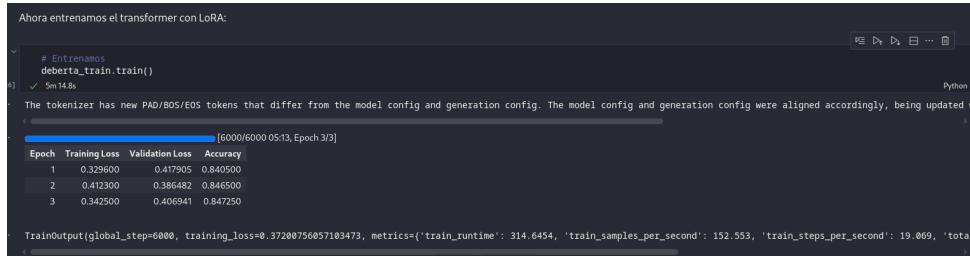
# Vemos cuanto vamos a entrenar
pesos_modelo_deberta = comparativa_modelos_finales[comparativa_modelos_finales['nombre'] == nombre_deberta]['parametros'].values
print(f'Parámetros totales: {pesos_modelo_deberta}')
print(f'Parámetros entrenables: {parametros_entrenar_deberta}')
print(f'Porcentaje de entreno: {(parametros_entrenar_deberta / pesos_modelo_deberta) * 100} %')

```

0.0s
Parámetros totales: [184423682]
Parámetros entrenables: 443986
Porcentaje de entreno: [0.24069902]

Figura 22: Configuración Deberta para este segundo intento.

Cuando lo ponemos a entrenar, vemos que ha bajado nuestro accuracy. Esto se debe ya que, normalmente, al vector k no se le suele aplicar esto al tener información de la posición (aunque no baja mucho tampoco, literal no se notan mucho los cambios). Miremos también el tiempo, entrenamos en 5 minutos, 1 minuto menos que antes (a lo mejor es porque ya estaba el modelo precargado, o cosa de nuestra RTX 4060).



Epoch	Training Loss	Validation Loss	Accuracy
1	0.329600	0.417905	0.840500
2	0.412300	0.386482	0.846500
3	0.342500	0.406941	0.847250

TrainOutput(global_step=6000, training_loss=0.37200756057103473, metrics={'train_runtime': 314.6454, 'train_samples_per_second': 152.553, 'train_steps_per_second': 19.069, 'total': 6000})

Figura 23: Entrenamiento Deberta para este segundo intento.

Para salir de dudas (y confirmar lo que decimos, vamos a ver las métricas con esta configuración:

3.1.4. Evaluación del entrenamiento

Ahora, vamos a ver algunas métricas con classification_report:

```
# Predecimos las reseñas de test
pred_proba_deberta = deberta_train.predict(test_deberta_tokenizado) # Ya usamos esto como validacion, solo sacaremos mas metricas

# Lo convertimos a 1 o 0
pred_deberta = np.argmax(pred_proba_deberta.predictions, axis=1)

# Mostramos metricas mas avanzadas
metricas_deberta_eval = classification_report(y_true=test_deberta['label'], y_pred=pred_deberta)
print(metricas_deberta_eval)
```

	precision	recall	f1-score	support
0	0.84	0.86	0.85	2028
1	0.86	0.83	0.84	1972
accuracy			0.85	4000
macro avg	0.85	0.85	0.85	4000
weighted avg	0.85	0.85	0.85	4000

Figura 24: Evaluación Deberta para este segundo intento.

En comparación con nuestro primer intento, vemos que no ha cambiado nada, solo un poco la precisión para la clase 0 que aumentó un 0.1, pero podemos decir que son los mismos resultados.

```
# Tokenizamos las reseñas
reseñas_eval_token_deberta = reseñas_evaluar_hugging.map(tokenizar_datos_deberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_deberta = deberta_train.predict(reseñas_eval_token_deberta)

# Vemos los resultados
pred_eval_deberta = np.argmax(pred_proba_eval_deberta.predictions, axis=1)
metricas_deberta_eval_personalizado = classification_report(y_true=pred_proba_eval_deberta.label_ids, y_pred=pred_eval_deberta)
print(metricas_deberta_eval_personalizado)
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	2
1	1.00	1.00	1.00	2
accuracy			1.00	4
macro avg	1.00	1.00	1.00	4
weighted avg	1.00	1.00	1.00	4

Figura 25: Evaluación personalizada Deberta para este segundo intento.

Confirmamos esto al ver que a nuestras reseñas las acierta todas.

4.1.3. Tercer intento: LoRA con r=4, alpha=16, aplicado a Q, K y V, 3 epochs

Ahora si vamos a ver que pasa si metemos una matriz más pequeña, con r=4.

```

# Configuramos el LoRA
configuracion_lora_deberta = LoraConfig(
    r=4, # Es recomendable comenzar con 8
    lora_alpha=16, # Recomendable el doble o cuádruple que r
    target_modules=['query_proj', 'key_proj', 'value_proj'], # Se lo aplicamos a los vectores q y v
    lora_dropout=0.1, # El dropout de siempre
    bias='none',
    task_type=TaskType.SEQ_CLS
)

# Aplicamos LoRA al modelo
deberta_model_lora = get_peft_model(deberta_model, configuracion_lora_deberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_deberta = sum(parametro.numel() for parametro in deberta_model_lora.parameters() if parametro.requires_grad)

# Vemos cuantos vamos a entrenar
pesos_modelo_deberta = comparativa_modelos_finales['comparativa_modelos_finales']['nombre'] == nombre_deberta]['parametros'].values
print(f'Parámetros totales: {pesos_modelo_deberta}')
print(f'Parámetros entrenables: {parametros_entrenar_deberta}')
print(f'Porcentaje de entreno: ({parametros_entrenar_deberta} / pesos_modelo_deberta) * 100')

```

0.05

```

Parámetros totales: [184423682]
Parámetros entrenables: 222722
Porcentaje de entreno: [0.12076649]

```

Figura 26: Configuración de LORA Deberta para este tercer intento.

Cuando lo ponemos a entrenar, vemos que, en efecto, demora menos (4 minutos y 55 segundos aproximadamente), y legamos otra vez a un accuracy del 85 %.

```

Ahora entrenamos el transformer con LoRA:

# Entrenamos
deberta_train.train()
4m54s
The tokenizer has new PAD/BOS/EOS tokens that differ from the model config and generation config. The model config and generation config were aligned accordingly, being updated.
[5640/5640 04:53, Epoch 3/3]
Epoch Training Loss Validation Loss Accuracy
1 0.410300 0.364858 0.842553
2 0.322100 0.411907 0.847340
3 0.356800 0.390379 0.851064

TrainOutput(global_step=5640, training_loss=0.3907317964743215, metrics={'train_runtime': 293.9225, 'train_samples_per_second': 153.51, 'train_steps_per_second': 19.189, 'total'

```

Figura 27: Entrenamiento de Deberta para este tercer intento.

Para confirmar las buenas sensaciones, vamos a ver que tal las demás métricas:

3.1.4. Evaluación del entrenamiento

Ahora, vamos a ver algunas métricas con `classification_report`:

```
# Predecimos las reseñas de test
pred_proba_deberta = deberta_train.predict(test_deberta_tokenizado) # Ya usamos esto como validacion, solo sacaremos mas metricas

# Lo convertimos a 1 o 0
pred_deberta = np.argmax(pred_proba_deberta.predictions, axis=1)

# Mostramos metricas mas avanzadas
metricas_deberta_eval = classification_report(y_true=test_deberta['label'], y_pred=pred_deberta)
print(metricas_deberta_eval)
```

	precision	recall	f1-score	support
0	0.83	0.87	0.85	1838
1	0.87	0.83	0.85	1922
accuracy			0.85	3760
macro avg	0.85	0.85	0.85	3760
weighted avg	0.85	0.85	0.85	3760

Figura 28: Evaluación de Deberta para este tercer intento.

Vemos que seguimos igual, un 0.85 para prácticamente todo, solo que subio un poco a favor de las reseñas positivas (clase 1). Vimos que, hay veces que coge más reseñas positivas, por lo cual, es normal este cambio 'diminuto'.

```
# Tokenizamos las reseñas
reseñas_eval_token_deberta = reseñas_evaluar_hugging.map(tokenizar_datos_deberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_deberta = deberta_train.predict(reseñas_eval_token_deberta)

# Vemos los resultados
pred_eval_deberta = np.argmax(pred_proba_eval_deberta.predictions, axis=1)
metricas_deberta_eval_personalizado = classification_report(y_true=pred_proba_eval_deberta.label_ids, y_pred=pred_eval_deberta)
print(metricas_deberta_eval_personalizado)
```

	precision	recall	f1-score	support
0	0.50	0.50	0.50	2
1	0.50	0.50	0.50	2
accuracy			0.50	4
macro avg	0.50	0.50	0.50	4
weighted avg	0.50	0.50	0.50	4

Figura 29: Evaluación personalizada de Deberta para este tercer intento.

Pero cuidado aquí, ha fallado en nuestras reseñas personalizadas. Veremos en los 2 últimos intentos por qué pasa eso.

4.1.4. Cuarto intento: LoRA con r=4, alpha=32, aplicado a Q, K y V, 3 epochs

Vamos a subirle la 'concentración', que a lo mejor, necesitamos que preste mejor atención a nuestra especialización (ver que tenemos menos parámetros entrenables, tanto en este como en el anterior intento).

```

# Configuramos el LoRA
configuracion_lora_deberta = LoraConfig(
    r=4, # Es recomendable comenzar con 8
    lora_alpha=32, # Recomendable el doble o cuádruple que 16
    target_modules=['query_proj', 'key_proj', 'value_proj'], # Se lo aplicamos a los vectores q y v
    lora_dropout=0.1, # El dropout de siempre
    bias='none',
    task_type=TaskType.SEQ_CLS
)

# Aplicamos LoRA al modelo
deberta_model_lora = get_peft_model(deberta_model, configuracion_lora_deberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_deberta = sum(parametro.numel() for parametro in deberta_model_lora.parameters() if parametro.requires_grad)

# Vemos cuantos tenemos
pesos_modelo_deberta = comparativa_modelos_finales[comparativa_modelos_finales['nombre'] == nombre_deberta]['parametros'].values
print(f'Parámetros totales: {pesos_modelo_deberta}')
print(f'Parámetros entrenables: {parametros_entrenar_deberta}')
print(f'Porcentaje de entrenamiento: {(parametros_entrenar_deberta / pesos_modelo_deberta) * 100}')

... 0.0s
... Parámetros totales: [184423682]
Parámetros entrenables: 222722
Porcentaje de entrenamiento: [0.12076649]

```

Figura 30: Configuración LoRA de Deberta para este cuarto intento.

Cuando lo ponemos a entrenar, vemos que no pasamos del 84% de accuracy, pero no nos enfoquemos mucho en eso, y mejor prestemos atención a las métricas (el entrenamiento demoró más, unos 7 minutos aproximadamente, no sabemos muy bien el por qué :().

Epoch	Training Loss	Validation Loss	Accuracy
1	0.390300	0.404172	0.840750
2	0.426500	0.411558	0.847000
3	0.327200	0.415059	0.845000

Figura 31: Entrenamiento de Deberta para este cuarto intento.

Vemos en la imagen de abajo que, hemos bajado un poco los scores, pero aún así sigue haciendo bastante bien (un 0.84 de f1 ya es bastante bueno).

```
# Predecimos las reseñas de test
pred_proba_deberta = deberta_train.predict(test_deberta_tokenizado) # Ya usamos esto como validacion

# Lo convertimos a 1 o 0
pred_deberta = np.argmax(pred_proba_deberta.predictions, axis=1)

# Mostramos metricas mas avanzadas
metricas_deberta_eval = classification_report(y_true=test_deberta['label'], y_pred=pred_deberta)
print(metricas_deberta_eval)
✓ 21.3s

precision    recall   f1-score   support
          0       0.84      0.86      0.85     2028
          1       0.85      0.83      0.84     1972

accuracy                           0.84      4000
macro avg       0.85      0.84      0.84      4000
weighted avg    0.85      0.84      0.84      4000
```

Figura 32: Evaluación de Deberta para este cuarto intento.

Y nos podemos ir contentos, porque predice todas nuestras reseñas personalizadas el modelo, por lo cual, de momento, esta puede ser la solución ganador **entre resultados y eficiencia (menos parámetros a entrenar)**.

```
# Tokenizamos las reseñas
reseñas_eval_token_deberta = reseñas_evaluar_hugging.map(tokenizar_datos_deberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_deberta = deberta_train.predict(reseñas_eval_token_deberta)

# Vemos los resultados
pred_eval_deberta = np.argmax(pred_proba_eval_deberta.predictions, axis=1)
metricas_deberta_eval_personalizado = classification_report(y_true=pred_proba_eval_deberta.label_ids, y_pred=pred_eval_deberta)
print(metricas_deberta_eval_personalizado)
✓ 0.0s
Map: 100% [██████████] 4/4 [00:00<00:00, 336.24 examples/s]

precision    recall   f1-score   support
          0       1.00      1.00      1.00       2
          1       1.00      1.00      1.00       2

accuracy                           1.00       4
macro avg       1.00      1.00      1.00       4
weighted avg    1.00      1.00      1.00       4
```

Figura 33: Evaluación personalizada de Deberta para este cuarto intento.

4.1.5. Quinto y último intento: LoRA con r=4, alpha=32, aplicado a Q y V, 3 epochs

Vamos a hacer este último intento, evitando tocar el vector k, ya que si la gente no lo suele tocar, es mejor que nosotros también respetemos eso (aquí si entrenamos la menor cantidad de parámetros, unos 149k aproximadamente).

```

# Configuramos el LoRA
configuracion_lora_deberta = LoraConfig(
    r=4, # Es recomendable comenzar con 8
    lora_alpha=32, # Recomendable el doble o cuádruple que r
    target_modules=['query_proj', 'value_proj'], # Se lo aplicamos a los vectores q y v
    lora_dropout=0.1, # El dropout de siempre
    bias='none'
)
task_type=TaskType.SEQ_CLS

# Aplicamos LoRA al modelo
deberta_model_lora = get_peft_model(deberta_model, configuracion_lora_deberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_deberta = sum(parametro.numel() for parametro in deberta_model_lora.parameters() if parametro.requires_grad)

# Vemos cuanto vamos a entrenar
pesos_modelo_deberta = comparativa_modelos_finales[comparativa_modelos_finales['nombre'] == nombre_deberta]['parametros'].values
print(f'Parámetros totales: {pesos_modelo_deberta}')
print(f'Parámetros entrenables: {parametros_entrenar_deberta}')
print(f'Porcentaje de entrenamiento: {(parametros_entrenar_deberta / pesos_modelo_deberta) * 100}'')

[22]: ✓ 0.0s
... Parámetros totales: [184423682]
Parámetros entrenables: 148994
Porcentaje de entrenamiento: [0.08078897]

```

Figura 34: Configuración LoRA de Deberta para este último intento.

Vemos que al entrenar, no pasamos del 0.84 de accuracy, pero bueno, solo es para tener en cuenta (demoramos solo 4 minutos con 40 segundos aproximadamente, el menor tiempo hasta ahora).

```

# Entrenamos
deberta_train.train()
✓ 4m 43.7s

The tokenizer has new PAD/BOS/EOS tokens that differ from the model

[6000/6000 04:43, Epoch 3/3]



| Epoch | Training Loss | Validation Loss | Accuracy |
|-------|---------------|-----------------|----------|
| 1     | 0.393400      | 0.382730        | 0.837750 |
| 2     | 0.393300      | 0.388806        | 0.845000 |
| 3     | 0.394000      | 0.399562        | 0.845750 |


```

Figura 35: Entrenamiento de Deberta para este último intento.

Al ver las demás métricas, vemos que en verdad, tenemos para casi todo un 0.85 de scores, bastante mejor que antes.

```

# Predecimos las reseñas de test
pred_proba_deberta = deberta_train.predict(test_deberta_tokenizado) # Ya usamos esto como validacion, solo sacaremos mas metricas

# Lo convertimos a 1 o 0
pred_deberta = np.argmax(pred_proba_deberta.predictions, axis=1)

# Mostramos metricas mas avanzadas
metricas_deberta_eval = classification_report(y_true=test_deberta['label'], y_pred=pred_deberta)
print(metricas_deberta_eval)

```

	precision	recall	f1-score	support
0	0.84	0.86	0.85	2028
1	0.85	0.84	0.84	1972
accuracy	0.85	0.85	0.85	4000
macro avg	0.85	0.85	0.85	4000
weighted avg	0.85	0.85	0.85	4000

Figura 36: Evaluación de Deberta para este último intento.

Y las buenas noticias, es que para Deberta, **esta configuración es la ganadora**, ya que, **con pocos parámetros, el menor tiempo y sin tovar k**, logramos predecir bien nuestras reseñas personalizadas, y no bajamos del 0.85 de scores en los demás. Mantenemos la consistencia entre rendimiento y recursos. ¡Genial!

```

# Tokenizamos las reseñas
reseñas_eval_token_deberta = reseñas_evaluar_hugging.map(tokenizar_datos_deberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_deberta = deberta_train.predict(reseñas_eval_token_deberta)

# Vemos los resultados
pred_eval_deberta = np.argmax(pred_proba_eval_deberta.predictions, axis=1)
metricas_deberta_eval_personalizado = classification_report(y_true=pred_proba_eval_deberta.label_ids, y_pred=pred_eval_deberta)
print(metricas_deberta_eval_personalizado)

```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	2
1	1.00	1.00	1.00	2
accuracy			1.00	4
macro avg	1.00	1.00	1.00	4
weighted avg	1.00	1.00	1.00	4

Figura 37: Evaluación personalizada de Deberta para este último intento.

4.2. LoRA para Roberta

Al igual que deberta, se probará unos intentos para ver los resultados. Al ser muy similar, solo se mencionará los resultados obtenidos.

4.2.1. Primer intento: LoRA con r=8, alpha=32, aplicado a Q y V, 3 epochs

Aplicaremos la misma configuración inicial que Deberta, así que será algo parecido a lo anterior visto (entrenamos solo un 0.7 % de los parámetros, unos 890k aproximadamente).

```

# Configuramos el LoRA
configuracion_lora_roberta = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=['query', 'value'],
    lora_dropout=0.1,
    bias="none",
    task_type=TaskType.SEQ_CLS
)

# Aplicamos LoRA a roberta
roberta_model_lora = get_peft_model(roberta_model, configuracion_lora_roberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_roberta = sum(parametro.numel() for parametro in roberta_model_lora.parameters() if parametro.requires_grad)

# Vemos cuanto vamos a entrenar
pesos_modelo_roberta = comparativa_modelos_finales['nombre'][comparativa_modelos_finales['nombre'] == nombre_roberta]['parametros'].values
print(f'Parámetros totales: {pesos_modelo_roberta}')
print(f'Parámetros entrenables: {parametros_entrenar_roberta}')
print(f'Porcentaje de entreno: {(parametros_entrenar_roberta / pesos_modelo_roberta) * 100} %')

```

0.0s

Parámetros totales: [124647170]
Parámetros entrenables: 887042
Porcentaje de entreno: [0.71164231]

Figura 38: Configuración LoRA para Roberta para este primer intento.

Al ponerlo a entrenar, no superamos el 0.84 de accuracy, pero vamos a ver que tal va con las demás métricas.

Epoch	Training Loss	Validation Loss	Accuracy
1	0.409400	0.381370	0.832250
2	0.357800	0.374599	0.839750
3	0.306400	0.386319	0.839250

Figura 39: Entrenamiento para Roberta para este primer intento.

Al ver las demás métricas (imagen de abajo), vemos que lo hace fatal para la clase 1 (reseñas positivas), y para las negativas, tampoco es que lo haga muy bien. Aún así, vamos a ver que tal le va con nuestras reseñas.

	precision	recall	f1-score	support
0	0.51	1.00	0.68	2028
1	0.97	0.03	0.06	1972

	accuracy			
accuracy	0.74	0.51	0.37	4000
macro avg	0.74	0.52	0.37	4000
weighted avg	0.74	0.52	0.37	4000

Figura 40: Evaluación para Roberta para este primer intento.

Al ver los resultados, vemos que no predice ninguna de la clase 1, seguramente sea un 'null model', es decir, predecir la clase mayoritaria (o la media). Peor resultado de momento.

```
Ahora con las reseñas inventadas:

# Tokenizamos las reseñas
reseñas_eval_token_roberta = reseñas_evaluar_hugging.map(tokenizar_datos_roberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_roberta = roberta_train.predict(reseñas_eval_token_roberta)

# Vemos los resultados
pred_eval_roberta = np.argmax(pred_proba_eval_roberta.predictions, axis=1)
metricas_roberta_eval_personalizado = classification_report(y_true=pred_proba_eval_roberta.label_ids, y_pred=pred_eval_roberta)
print(metricas_roberta_eval_personalizado)
[✓] 0.2s
Map: 100% [██████████] 4/4 [00:00<00:00, 187.53 examples/s]

precision    recall    f1-score   support
          0       0.50      1.00      0.67       2
          1       0.00      0.00      0.00       2

   accuracy        0.50       4
macro avg       0.25      0.50      0.33       4
weighted avg    0.25      0.50      0.33       4
```

Figura 41: Evaluación personalizada para Roberta para este primer intento.

4.2.2. Segundo intento: LoRA con r=8, alpha=32, aplicado a Q, K y V, 3 epochs

Aquí hicimos lo mismo: Por curiosidad, probamos todos los vectores, a ver que tal (1.03M de parámetros aproximadamente).

```
# Configuramos el LoRA
configuracion_lora_roberta = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=['query', 'key', 'value'],
    lora_dropout=0.1,
    bias="none",
    task_type=TaskType.SEQ_CLS
)

# Aplicamos LoRA a roberta
roberta_model_lora = get_peft_model(roberta_model, configuracion_lora_roberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_roberta = sum(parametro.numel() for parametro in roberta_model_lora.parameters() if parametro.requires_grad)

# Vemos cuantos vamos a entrenar
pesos_modelo_roberta = comparativa_modelos_finales[comparativa_modelos_finales['nombre'] == nombre_roberta]['parametros'].values
print(f'Parámetros totales: {pesos_modelo_roberta}')
print(f'Parámetros entrenables: {parametros_entrenar_roberta}')
print(f'Porcentaje de entrenamiento: {(parametros_entrenar_roberta / pesos_modelo_roberta) * 100}')
[✓] 0.0s
Parámetros totales: [124647170]
Parámetros entrenables: 1034498
Porcentaje de entrenamiento: [0.82994182]
```

Figura 42: Configuración LoRA para Roberta para este segundo intento.

Al entrenarlo, ha subido algo el accuracy (a 0.84), así que es un buen paso (solo entrenó por 3 minutos, similar al primer intento).

```
# Entrenamiento de roberta
# Entrenamos
roberta_train.train()
[✓] 3m & 1s
[6000/6000 03:07, Epoch 3/3]

Epoch Training Loss Validation Loss Accuracy
1 0.410400 0.475229 0.831250
2 0.423800 0.389889 0.835250
3 0.378500 0.416996 0.842000

TrainOutput(global_step=6000, training_loss=0.402559930096663, metrics={'train_runtime': 188.0325, 'train_samples_per_second': 255.275, 'train_steps_per_second': 31.909, 'total_']}
```

Figura 43: Entrenamiento LoRA para Roberta para este segundo intento.

Al evaluarlo, vemos que mejoró una barbaridad los scores (0.84 en todo):

3.2.2. Evaluacion del modelo

Vamos a ver si funciona o no, viendo métricas como antes lo hicimos con deberta:

```
# Predecimos las reseñas de test
pred_proba_roberta = roberta_train.predict(test_roberta_tokenizado) # Ya usamos esto como validacion, solo sacaremos mas metricas

# Lo convertimos a 1 o 0
pred_roberta = np.argmax(pred_proba_roberta.predictions, axis=1)

# Mostamos metricas mas avanzadas
metricas_roberta_eval = classification_report(y_true=test_roberta['label'], y_pred=pred_roberta)
print(metricas_roberta_eval)
```

precision recall f1-score support

	0	1		
0	0.84	0.86	0.85	2028
1	0.85	0.83	0.84	1972
accuracy			0.84	4000
macro avg	0.84	0.84	0.84	4000
weighted avg	0.84	0.84	0.84	4000

Figura 44: Evaluación para Roberta para este segundo intento.

Y también mejoró con nuestras reseñas, pero aún así, no nos basta (queremos igualar a Deberta):

Ahora con las reseñas inventadas:

```
# Tokenizamos las reseñas
reseñas_eval_token_roberta = reseñas_evaluar_hugging.map(tokenizar_datos_roberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_roberta = roberta_train.predict(reseñas_eval_token_roberta)

# Vemos los resultados
pred_eval_roberta = np.argmax(pred_proba_eval_roberta.predictions, axis=1)
metricas_roberta_eval_personalizado = classification_report(y_true=pred_proba_eval_roberta.label_ids, y_pred=pred_eval_roberta)
print(metricas_roberta_eval_personalizado)
```

Map: 100% 4/4 [00:00<00:00, 692.64 examples/s]

	0	1		
precision	1.00	0.50	0.67	2
recall	0.67	1.00	0.80	2
f1-score			0.75	4
support			4	
accuracy			0.83	4
macro avg	0.83	0.75	0.73	4
weighted avg	0.83	0.75	0.73	4

Figura 45: Evaluación personalizada para Roberta para este segundo intento.

4.2.3. Tercer intento: LoRA con r=4, alpha=16, aplicado a Q, K y V, 3 epochs

Ahora, haremos el intento de mejorar los resultados, pero con menos parámetros (813K parámetros aproximadamente).

```

# Configuramos el LoRA
configuracion_lora_roberta = LoraConfig(
    r=4,
    lora_alpha=16,
    target_modules=['query', 'key', 'value'],
    lora_dropout=0.1,
    bias='none',
    task_type=TaskType.SEQ_CLS
)

# Aplicamos LoRA a roberta
roberta_model_lora = get_peft_model(roberta_model, configuracion_lora_roberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_roberta = sum(parametro.numel() for parametro in roberta_model_lora.parameters() if parametro.requires_grad)

# Vemos cuanto vamos a entrenar
pesos_modelo_roberta = comparativa_modelos_finales[comparativa_modelos_finales['nombre'] == nombre_roberta]['parametros'].values
print(f'Parámetros totales: {pesos_modelo_roberta}')
print(f'Parámetros entrenables: {parametros_entrenar_roberta}')
print(f'Porcentaje de entreno: {(parametros_entrenar_roberta / pesos_modelo_roberta) * 100}')

```

0.0s

Parámetros totales: [124647170]
Parámetros entrenables: 813314
Porcentaje de entreno: [0.65249295]

+ Code + Markdown

Figura 46: Configuración LoRA para Roberta para este tercer intento.

Al entrenarlo, sigue siendo extremadamente más rápido (3 min aproximadamente), y el accuracy increíblemente ha mejorado.

```

# Entrenamiento de roberta
# Entrenamos
roberta_train.train()

```

3m1.4s

Epoch	Training Loss	Validation Loss	Accuracy
1	0.400400	0.382117	0.835904
2	0.350000	0.423565	0.844149
3	0.354200	0.402634	0.842819

TrainOutput(global_step=5640, training_loss=0.407526636800022, metrics={'train_runtime': 181.3311, 'train_samples_per_second': 248.827, 'train_steps': 5640})

+ Code + Markdown

Figura 47: Entrenamiento para Roberta para este tercer intento.

Viendo las métricas, vemos que sigue casi igual, con la diferencia que bajo un poco el score para la clase 1, pero mejoró para la clase 0.

3.2.2. Evaluacion del modelo

Vamos a ver si funciona o no, viendo métricas como antes lo hicimos con deberta:

```
# Predecimos las reseñas de test
pred_proba_roberta = roberta_train.predict(test_roberta_tokenizado) # Ya usamos esto como validacion, solo sacaremos mas metricas

# Lo convertimos a 1 o 0
pred_roberta = np.argmax(pred_proba_roberta.predictions, axis=1)

# Mostramos metricas mas avanzadas
metricas_roberta_eval = classification_report(y_true=test_roberta['label'], y_pred=pred_roberta)
print(metricas_roberta_eval)
7] ✓ 5.2s

precision    recall   f1-score   support
          0       0.81      0.88      0.85     1838
          1       0.88      0.81      0.84     1922

accuracy                           0.84      3760
macro avg                           0.84      0.84      0.84     3760
weighted avg                          0.85      0.84      0.84     3760
```

Figura 48: Evaluación para Roberta para este tercer intento.

Y para nuestras reseñas personalizadas, aún no logramos mejorar los resultados del intento anterior (algo de esperar).

```
Ahora con las reseñas inventadas:

# Tokenizamos las reseñas
reseñas_eval_token_roberta = reseñas_evaluar_hugging.map(tokenizar_datos_roberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_roberta = roberta_train.predict(reseñas_eval_token_roberta)

# Vemos los resultados
pred_eval_roberta = np.argmax(pred_proba_eval_roberta.predictions, axis=1)
metricas_roberta_eval_personalizado = classification_report(y_true=pred_proba_eval_roberta.label_ids, y_pred=pred_eval_roberta)
print(metricas_roberta_eval_personalizado)
8] ✓ 0.0s

Map: 100% [██████████] 4/4 [00:00<00:00, 664.15 examples/s]

precision    recall   f1-score   support
          0       1.00      0.50      0.67       2
          1       0.67      1.00      0.80       2

accuracy                           0.75       4
macro avg                           0.83      0.75      0.73       4
weighted avg                          0.83      0.75      0.73       4
```

Figura 49: Evaluación personalizada para Roberta para este tercer intento.

4.2.4. Cuarto intento: LoRA con r=4, alpha=32, aplicado a Q, K y V, 3 epochs

Vamos a aumentar la concentración, que fue eso lo que solucionó el problema con Deberta (mismos parámetros que el anterior intento).

```

# Configuramos el LoRA
configuracion_lora_roberta = LoraConfig(
    r=4,
    lora_alpha=32,
    target_modules=['query', 'key', 'value'],
    lora_dropout=0.1,
    bias="none",
    task_type=TaskType.SEQ_CLS
)

# Aplicamos LoRA a roberta
roberta_model_lora = get_peft_model(roberta_model, configuracion_lora_roberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_roberta = sum(parametro.numel() for parametro in roberta_model_lora.parameters() if parametro.requires_grad)

# Vemos cuanto vamos a entrenar
pesos_modelo_roberta = comparativa_modelos_finales[comparativa_modelos_finales['nombre'] == nombre_roberta]['parametros'].values
print(f'Parámetros totales: {pesos_modelo_roberta}')
print(f'Parámetros entrenables: {parametros_entrenar_roberta}')
print(f'Porcentaje de entreno: {((parametros_entrenar_roberta / pesos_modelo_roberta) * 100)}')
print(f'0.0s')

Parámetros totales: [124647170]
Parámetros entrenables: 813314
Porcentaje de entreno: [0.65249295]

```

Figura 50: Configuración LoRA para Roberta para este cuarto intento.

Al entrenar el modelo, bajo un poco el accuracy, pero sabemos que este no es el veredicto final (entrenó 4 minutos, 1 minuto más que antes).

```

# Entrenamiento de roberta
# Entrenamos
roberta_train.train()

```

✓ 4m 40.8s

Epoch	Training Loss	Validation Loss	Accuracy
1	0.413500	0.423968	0.833500
2	0.423000	0.420411	0.838000
3	0.338800	0.420015	0.841000

[6000/6000 04:40, Epoch 3/3]

TrainOutput(global_step=6000, training_loss=0.4057218359311422, metrics={'train_1': 0.841})

Figura 51: Entrenamiento para Roberta para este cuarto intento.

Viendo las métricas, ahora está más parejo entre las clases 0 y 1, y seguimos con un score general de 0.84.

3.2.2. Evaluacion del modelo

Vamos a ver si funciona o no, viendo métricas como antes lo hicimos con deberta:

```
# Predecimos las reseñas de test
pred_proba_roberta = roberta_train.predict(test_roberta_tokenizado) # Ya usamos esto como validacion, solo sacaremos mas metricas

# Lo convertimos a 1 o 0
pred_roberta = np.argmax(pred_proba_roberta.predictions, axis=1)

# Mostramos metricas mas avanzadas
metricas_roberta_eval = classification_report(y_true=test_roberta['label'], y_pred=pred_roberta)
print(metricas_roberta_eval)
```

	precision	recall	f1-score	support
0	0.83	0.86	0.85	2028
1	0.85	0.82	0.84	1972
accuracy			0.84	4000
macro avg	0.84	0.84	0.84	4000
weighted avg	0.84	0.84	0.84	4000

Figura 52: Evaluación para Roberta para este cuarto intento.

Y viendo los resultados con nuestras reseñas... ¡Lo hemos conseguido! Con menos pesos que el primer intento, hemos conseguido un modelo que puede moverse bien con casi cualquier reseña.

Ahora con las reseñas inventadas:

```
# Tokenizamos las reseñas
reseñas_eval_token_roberta = reseñas_evaluar_hugging.map(tokenizar_datos_roberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_roberta = roberta_train.predict(reseñas_eval_token_roberta)

# Vemos los resultados
pred_eval_roberta = np.argmax(pred_proba_eval_roberta.predictions, axis=1)
metricas_roberta_eval_personalizado = classification_report(y_true=pred_proba_eval_roberta.label_ids, y_pred=pred_eval_roberta)
print(metricas_roberta_eval_personalizado)
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	2
1	1.00	1.00	1.00	2
accuracy			1.00	4
macro avg	1.00	1.00	1.00	4
weighted avg	1.00	1.00	1.00	4

Figura 53: Evaluación personalizada para Roberta para este cuarto intento.

4.2.5. Quinto y último intento: LoRA con r=4, alpha=32, aplicado a Q y V, 3 epochs

Por último, intentaremos aplicar LoRA solo a Q y V, y no a K (y de paso, entrenamos menos parámetros, en total, unos 740K).

```

> ~
# Configuramos el LoRA
configuracion_lora_roberta = LoraConfig(
    r=4,
    lora_alpha=32,
    target_modules=['query', 'value'],
    lora_dropout=0.1,
    bias="none",
    task_type=TaskType.SEQ_CLS
)

# Aplicamos LoRA a roberta
roberta_model_lora = get_peft_model(roberta_model, configuracion_lora_roberta)

# Obtenemos cuantos parametros vamos a entrenar
parametros_entrenar_roberta = sum(parametro.numel() for parametro in roberta_model_lora.parameters() if parametro.requires_grad)

# Vemos cuanto vamos a entrenar
pesos_modelo_roberta = comparativa_modelos_finales[comparativa_modelos_finales['nombre'] == nombre_roberta]['parametros'].values
print(f'Parámetros totales: {pesos_modelo_roberta}')
print(f'Parámetros entrenables: {parametros_entrenar_roberta}')
print(f'Porcentaje de entrenamiento: {(parametros_entrenar_roberta / pesos_modelo_roberta) * 100}')
33] ✓ 0.0s
.. Parámetros totales: [124647170]
Parámetros entrenables: 739586
Porcentaje de entrenamiento: [0.5933436]

```

Figura 54: Configuración LoRA para Roberta para este último intento.

Al entrenar, es verdad que hemos bajado un poco el accuracy (0.83), pero mejor veamos las métricas, que son nuestro veredicto final (entreno de 2 minutos, lo más bajo hasta ahora).

```

# Entrenamiento de roberta
# Entrenamos
roberta_train.train()
5] ✓ 3m 7.4s
[6000/6000 03:07, Epoch 3/3]


| Epoch | Training Loss | Validation Loss | Accuracy |
|-------|---------------|-----------------|----------|
| 1     | 0.445700      | 0.409155        | 0.823500 |
| 2     | 0.474800      | 0.413340        | 0.834250 |
| 3     | 0.432100      | 0.411126        | 0.833750 |


TrainOutput(global_step=6000, training_loss=0.40658024708429974, n

```

Figura 55: Entrenamiento para Roberta para este último intento.

Viendo las métricas, está muy similar al cuarto intento, solo que más parejo para ambas clases (0.84).

3.2.2. Evaluacion del modelo

Vamos a ver si funciona o no, viendo métricas como antes lo hicimos con deberta:

```
# Predecimos las reseñas de test
pred_proba_roberta = roberta_train.predict(test_roberta_tokenizado) # Ya usamos esto como validacion, solo sacaremos mas metricas

# Lo convertimos a 1 o 0
pred_roberta = np.argmax(pred_proba_roberta.predictions, axis=1)

# Mostramos metricas mas avanzadas
metricas_roberta_eval = classification_report(y_true=test_roberta['label'], y_pred=pred_roberta)
print(metricas_roberta_eval)
```

[2/500 00:00 < 00:04, 102.24 it/s]

	precision	recall	f1-score	support
0	0.84	0.85	0.84	2028
1	0.84	0.83	0.83	1972
accuracy			0.84	4000
macro avg	0.84	0.84	0.84	4000
weighted avg	0.84	0.84	0.84	4000

Figura 56: Evaluación para Roberta para este último intento.

Y viendo nuestras reseñas personalizadas, es algo raro: Hay veces que nos dió un perfecto, y a veces nos dió un resultado similar al cuarto intento. Como recomendación, para un hardware pequeño, el **cuarto intento es la opción más fiable, mientras que el quinto, esta bien pero no hay tanta 'confianza' como con el cuarto intento.**

Ahora con las reseñas inventadas:

```
# Tokenizamos las reseñas
reseñas_eval_token_roberta = reseñas_evaluar_huggingface(tokenizar_datos_roberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_roberta = roberta_train.predict(reseñas_eval_token_roberta)

# Vemos los resultados
pred_eval_roberta = np.argmax(pred_proba_eval_roberta.predictions, axis=1)
metricas_roberta_eval_personalizado = classification_report(y_true=reseñas_eval_token_roberta.label_ids, y_pred=pred_eval_roberta)
print(metricas_roberta_eval_personalizado)
```

Map: 100% 444 [00:00:00.00, 637.09 examples]

	precision	recall	f1-score	support
0	1.00	1.00	1.00	2
1	1.00	1.00	1.00	2
accuracy			1.00	4
macro avg	1.00	1.00	1.00	4
weighted avg	1.00	1.00	1.00	4

(a) Evaluación personalizada Roberta para este último intento (acierto).

```
# Tokenizamos las reseñas
reseñas_eval_token_roberta = reseñas_evaluar_huggingface(tokenizar_datos_roberta, batched=True, remove_columns='reviews')

# Se lo pasamos al modelo
pred_proba_eval_roberta = roberta_train.predict(reseñas_eval_token_roberta)

# Vemos los resultados
pred_eval_roberta = np.argmax(pred_proba_eval_roberta.predictions, axis=1)
metricas_roberta_eval_personalizado = classification_report(y_true=reseñas_eval_token_roberta.label_ids, y_pred=pred_eval_roberta)
print(metricas_roberta_eval_personalizado)
```

Map: 100% 444 [00:00:00.00, 634.13 examples]

	precision	recall	f1-score	support
0	1.00	0.50	0.67	2
1	0.67	1.00	0.80	2
accuracy			0.75	4
macro avg	0.83	0.75	0.73	4
weighted avg	0.83	0.75	0.73	4

(b) Evaluación personalizada Roberta para este último intento (fallo).

Figura 57: Resultados de la evaluación personalizada para Roberta en este último intento, muy diferentes dependiendo de la ejecución.

5. Comparación y justificación de resultados

Vamos a comparar los resultados de los mejores intentos, ya que, no vale la pena comparar resultados de intentos que no fueron tan buenos.

En general, se vió que ambos modelos lo hicieron bastante bien. Si vemos los resultados del **último intento de Deberta** y el **último intento de Roberta** (al final, supondremos que fue una cosa extraña que pasó con los resultados), vemos resultados similares.

```

print('=' * 15, 'Metricas para deberta', '=' * 15)
print(metricas_deberta_eval)
print('=' * 15, 'Metricas para roberta', '=' * 15)
print(metricas_roberta_eval)

✓ 0.0s

===== Metricas para deberta =====
precision    recall   f1-score   support
0            0.84      0.86      0.85      2028
1            0.85      0.84      0.84      1972

accuracy          0.85      0.85      0.85      4000
macro avg        0.85      0.85      0.85      4000
weighted avg     0.85      0.85      0.85      4000

===== Metricas para roberta =====
precision    recall   f1-score   support
0            0.84      0.85      0.84      2028
1            0.84      0.83      0.83      1972

accuracy          0.84      0.84      0.84      4000
macro avg        0.84      0.84      0.84      4000
weighted avg     0.84      0.84      0.84      4000

```

Figura 58: Comparación de métricas de Deberta y Roberta.

Sale ganador Deberta por tener un mejor (diminuto) score general con las reseñas de test, pero Roberta lo hace igual de bien. Veamos la comparación de parámetros/pesos entrenables:

```
Vemos que empatan en f1 ambos modelos (tanto en f1 macro como weighted, aunque este último no importa mucho al estar tan cerca). Deberata tiene un poquito mejor f1, aunque insignificante. Lo principal es que, partimos que deberata tiene más parámetros que roberta.
```

```
# Imprimimos el numero de parametros entrenados por cada uno
print('=' * 15, 'Parametros deberata', '=' * 15)
print(f'Parametros entrenables: {parametros_entrenar_deberta}')
print(f'Porcentaje de entreno: {((parametros_entrenar_deberta / pesos_modelo_deberta) * 100)}')

print('=' * 15, 'Parametros roberta', '=' * 15)
print(f'Parametros entrenables: {parametros_entrenar_roberta}')
print(f'Porcentaje de entreno: {((parametros_entrenar_roberta / pesos_modelo_roberta) * 100)}')
✓ 0.0s
=====
Parametros deberata =====
Parametros entrenables: 148994
Porcentaje de entreno: [0.08078897]
=====
Parametros roberta =====
Parametros entrenables: 739586
Porcentaje de entreno: [0.5933436]
```

+ Code + Markdown

Figura 59: Comparación de parámetros de Deberta y Roberta.

Aquí, para gente con poco/limitado hardware, sale ganador de nuevo Deberta, al entrenar muy pocos parámetros en general (tanto en porcentaje como en números). Teníamos en cuenta que Deberta tiene más parámetros que Roberta, esta puede ser una de las razones.

Por último, veamos qué pasa con las métricas de nuestras reseñas:

```
▷ 
    print('=' * 15, 'Metricas para deberata', '=' * 15)
    print(metricas_deberta_eval_personalizado)
    print('=' * 15, 'Metricas para roberta', '=' * 15)
    print(metricas_roberta_eval_personalizado)
[40] ✓ 0.0s
...
=====
Metricas para deberata =====
precision      recall   f1-score   support
0            1.00      1.00      1.00       2
1            1.00      1.00      1.00       2

accuracy                      1.00       4
macro avg                     1.00      1.00      1.00       4
weighted avg                  1.00      1.00      1.00       4

=====
Metricas para roberta =====
precision      recall   f1-score   support
0            1.00      0.50      0.67       2
1            0.67      1.00      0.80       2

accuracy                      0.75       4
macro avg                     0.83      0.75      0.73       4
weighted avg                  0.83      0.75      0.73       4
```

Figura 60: Comparación de la evaluación personalizada de Deberta y Roberta.

Aquí hay doble barra de medir: Roberta a veces falló, pero hay veces que acertó, mientras que Deberta lo hizo perfecto. Investigando un poco, y gracias a la ayuda de Gemini, encontramos esto:

Nota importante

Para Deberta, el mecanismo de atención de BERT/RoBERTa tiene un fallo matemático. Vamos a arreglarlo. Para Roberta, BERT estaba mal entrenado. Vamos a entrenarlo bien.

Esta diferenciación es clave, ya que, hace que Deberta arregle un problema en los cálculos (Microsoft introdujo la Atención Desempaquetada (Disentangled Attention)). En pocas palabras, separa el contenido de la posición (es algo un poco raro). Vemos que no se quivocan del todo, ya que Deberta, en general, presentó mejores resultados que Roberta. Viendo antes que también, la única desventaja de Deberta era su tiempo de entrenamiento (más que Roberta), sus resultados, y su idea de arreglar ese desajuste matemático introduciendo la atención desempaquetada, hace que los resultados de Deberta sean mejores que Roberta. Como conclusión final, Deberta es nuestro ganador frente a Roberta.

6. Extra: Comparación con PROMPT-ENGINEERING

Ya sabemos que Deberta es mejor ante Roberta. Pero, ¿Será mejor que utilizar técnicas de prompt engineering? ¿Ganará un zero/few shot frente a LoRA? Vamos a realizar esta pequeña comparativa para salir de dudas.

6.1. Búsqueda de modelos generativos

Vamos a buscar un modelo decoder-only, que esté especializado en **text-generation**, que sepan el lenguaje inglés, y los ordenamos por descargas:

```

# Modelo que vamos a usar
mejores_modelos_decoder = list(
    list_models(
        filter=['text-generation', 'en'],
        sort='downloads',
        direction=-1,
        limit=10
    )
)

# Sacamos informacion importante
info_modelos_genrativos = {
    'nombre': [modelo.id for modelo in mejores_modelos_decoder],
    'autor': [modelo.author for modelo in mejores_modelos_decoder],
    'descargas': [modelo.downloads for modelo in mejores_modelos_decoder],
    'funcion': [modelo.pipeline_tag for modelo in mejores_modelos_decoder],
    'tags': [modelo.tags for modelo in mejores_modelos_decoder]
}

# Mostramos en un dataframe
df_modelos_decoder = pd.DataFrame(info_modelos_genrativos)
df_modelos_decoder

```

	nombre	autor	descargas	funcion	tags
0	openai-community/gpt2	None	9934564	text-generation	[transformers, pytorch, tf, jax, tflite, rust, ...]
1	Qwen/Qwen2.5-7B-Instruct	None	7812528	text-generation	[transformers, safetensors, qwen2, text-genera...]
2	Qwen/Qwen2.5-3B-Instruct	None	6100885	text-generation	[transformers, safetensors, qwen2, text-genera...]
3	meta-llama/Llama-3.1-BB-Instruct	None	5221966	text-generation	[transformers, safetensors, llama, text-genera...]
4	Qwen/Qwen2.5-1.5B-Instruct	None	5087315	text-generation	[transformers, safetensors, qwen2, text-genera...]
5	facebook/opt-125m	None	4492941	text-generation	[transformers, pytorch, tf, jax, opt, text-gen...]
6	meta-llama/Llama-3.2-1B-Instruct	None	3686452	text-generation	[transformers, safetensors, llama, text-genera...]
7	Gensyn/Qwen2.5-0.5B-Instruct	None	3572773	text-generation	[transformers, safetensors, qwen2, text-genera...]
8	meta-llama/Llama-3.2-1B	None	3275581	text-generation	[transformers, safetensors, llama, text-genera...]
9	Qwen/Qwen2.5-Coder-0.5B-Instruct	None	3142890	text-generation	[transformers, safetensors, qwen2, text-genera...]

Figura 61: Lista de mejores modelos generativos en inglés.

Según ChatGPT, **Qwen2.5-7B-Instruct** es el mejor modelo para aplicarle prompt-engineering (sin usar los vistos en clase como gpt-2).

Nota importante

Mejor equilibrio (para reseñas coloquiales con prompt engineering): Qwen2.5-7B-Instruct → potencia y buena comprensión de inglés coloquial.

6.2. Zero-shot

Para este caso, zero-shot consiste en **no darle ejemplos al modelo, y decirle lo que queremos que haga**. Para ello, cogeremos solo algunas reseñas, al ser demasiadas (y el modelo demora en responder, a parte que a veces no te escupe exáctamente lo que quieres):

```
# Vamos a obtener las reseñas
reseñas_usar = reviews_dataframe['reviews'].iloc[:60] # Solo 60 porque sino se tarda mucho
label = reviews_dataframe['label'].iloc[:60]

Vamos a configurarlo para poder usarlo.

# Configuraremos el modelo
zero_shot = pipeline(
    model=modelo_usar,
    tokenizer=token_decoder,
    task='text-generation',
    device=-1
)

# Configuraremos el modelo
configuración_zero_shot = {
    'max_new_tokens': 45,
    'temperature': 0.05,
    'top_k': 10,
    'top_p': 0.8
}

Could not render content for 'application/vnd.jupyter.widget-view+json'
{"model_id": "678b55558f2b4d2c8c9788df67a094f5", "version_major": 2, "version_minor": 0}

Device set to use cpu
```

Figura 62: Reseñas a usar y configuración del modelo generativo.

Luego de pasarle un prompt (es gigante, para ello, ver el notebook), vemos que esta técnica no funciona muy bien: Obtenemos scores bajos, no logrando pasar el 0.75, que es lo que al menos, nosotros buscamos.

```
Nuevas predicciones del chunk 0: [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0]
Nuevas predicciones del chunk 15: [0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0]
Nuevas predicciones del chunk 30: [0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1]
Nuevas predicciones del chunk 45: [1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1]
[0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1

Vamos a ver como le fue:
```

```
print(classification_report(y_true=label, y_pred=resultados))

      precision    recall  f1-score   support

          0       0.64      0.83      0.72       30
          1       0.76      0.53      0.63       30

   accuracy                           0.68       60
  macro avg       0.70      0.68      0.68       60
weighted avg       0.70      0.68      0.68       60
```

Figura 63: Métricas zero-shot.

Y si pasamos nuestras reseñas, no lo hace mal, pero no son muy buenas (comparable a los intentos malos con LoRA de ambos modelos):

```

    # Pedimos respuesta al modelo
    lista_prediccion_zero_shot = obtener_predicciones_chunk_correctas(reseñas_evaluar['reviews'])

    # Mostramos el resultado final
    print(lista_prediccion_zero_shot)

    # Mostramos las metricas
    print(classification_report(y_true=reseñas_evaluar['label'], y_pred=lista_prediccion_zero_shot))
]

[1, 0, 1, 1]
      precision    recall   f1-score   support
          0       1.00     0.50     0.67      2
          1       0.67     1.00     0.80      2

      accuracy                           0.75      4
   macro avg       0.83     0.75     0.73      4
weighted avg       0.83     0.75     0.73      4

```

Figura 64: Métricas con las reseñas personalizadas zero-shot.

6.3. Few-shot

Aquí, es parecido a zero-shot, con la diferencia que **pasamos algunos ejemplos al modelo generativo, para que sepa como clasificarlos**. Nosotros decidimos cuantos ejemplos pasarles, pero comenzaremos con 5 porque lo hace bastante bien:

5.2. Few-shot

Vamos a aplicar la técnica de prompt engineering (poner algunos ejemplos), y veremos que tal le va contra el dataset y las reseñas inventadas por nosotros:

```

# Sacamos algunos ejemplos
n_samples = 5 # Esto se puede cambiar
examples = reviews_dataframe.iloc[:n_samples, :]

# Lo pasamos a dict
examples_few_shot = examples.to_dict(orient="records")
examples_few_shot

```

```

[{"reviews": "I lost all my gameplay after the update was released",
 'label': 1},
 {"reviews": "bad stupiud baD, O;PTION FOR CHECKPOINTS (SPOILER: THERE IS NON) NOT ADVISED FOR TWITCHY FINGERED PLAYERS OR YOU WILL DIE",
 'label': 0},
 {"reviews": "i came to this game many times and now my keyboard stopped working jerking jason out...",
 'label': 1},
 {"reviews": "shit fuck this game", 'label': 0},
 {"reviews": "BEST GAME", 'label': 1}]

```

Figura 65: Ejemplos que se le pasará al modelo.

Solo nos queda repetir lo mismo, añadiendo al prompt los ejemplos (ver el notebook que ahí se mira mejor). Los resultados, a decir verdad, son mucho mejores, pero hasta el peor modelo Roberta fine-tuneado con LoRA logra superar al modelo (sin decir que, en general, demora en dar las respuestas, también porque usamos la CPU porque no nos cabe en la GPU):

```
Nuevas predicciones del chunk 0: [0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0]
Nuevas predicciones del chunk 15: [0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0]
Nuevas predicciones del chunk 30: [1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1]
Nuevas predicciones del chunk 45: [1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1]
[0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1]

Vemos que tal le fue:
```

```
print(classification_report(y_true=label, y_pred=resultado))

      precision    recall  f1-score   support
0       0.71      0.83      0.77      30
1       0.80      0.67      0.73      30

accuracy                           0.75      60
macro avg       0.76      0.75      0.75      60
weighted avg    0.76      0.75      0.75      60
```

Figura 66: Resultados y métricas con few-shot.

Y con nuestras reseñas personalizadas, mejoró en el sentido que predice casi bien ambas clases, pero aún así, lejos del rendimiento de LoRA:

```
# Pedimos respuesta al modelo
lista_prediccion_zero_shot = obtener_predicciones_chunk_correctas(reseñas_evaluar['reviews'])

# Mostramos el resultado final
print(lista_prediccion_zero_shot)

# Mostramos las métricas
print(classification_report(y_true=reseñas_evaluar['label'], y_pred=lista_prediccion_zero_shot))
]

[1, 0, 1, 1]
      precision    recall  f1-score   support
0       1.00      0.50      0.67      2
1       0.67      1.00      0.80      2

accuracy                           0.75      4
macro avg       0.83      0.75      0.73      4
weighted avg    0.83      0.75      0.73      4
```

Figura 67: Resultados y métricas con las reseñas personalizadas para few-shot.

6.4. Conclusiones de esta comparación

Como conclusión general, no está mal las técnicas de prompt engineering (por algo, es una ingeniería aparte), pero LoRA, viendo zero y few-shot, gana casi por goleada en muchos aspectos (menos en pesos, ya que no entrenamos ninguno). No podemos decir nada de prompt engineering ni que no funciona, ya que vimos solo las técnicas **más básicas de esta ingeniería**. Seguramente, probando más técnicas como 'chain of thought', 'tree of thought', 'RA (reason and act)' y muchas otras más (como Persona, rol, etc), seguramente estaríamos en una comparación más pareja entre LoRA y prompt engineering. Aún así,

para estos casos, es mejor LoRA ya que podemos personalizar los modelos como queramos (en nuestra opinión), por algo es que muchos fine-tunning son con LoRA (aparte que la idea es muy buena y está chida).



Figura 68: Imagen generada con Gemini, prompt engineering vs LoRA.

7. RAG: Retrieval and Augmented Generation

Esta técnica consiste en darle a los modelos información actual (o personalizada) de alguna manera, para que puedan responder con información que no tenían a la hora de ser entrenados. Veremos nuestro caso de estudio y detallaremos como usamos RAG comparando 2 modelos 'distintos' (si, son distintos, pero en cuanto a pre-entrenamiento/datasets/especialización).

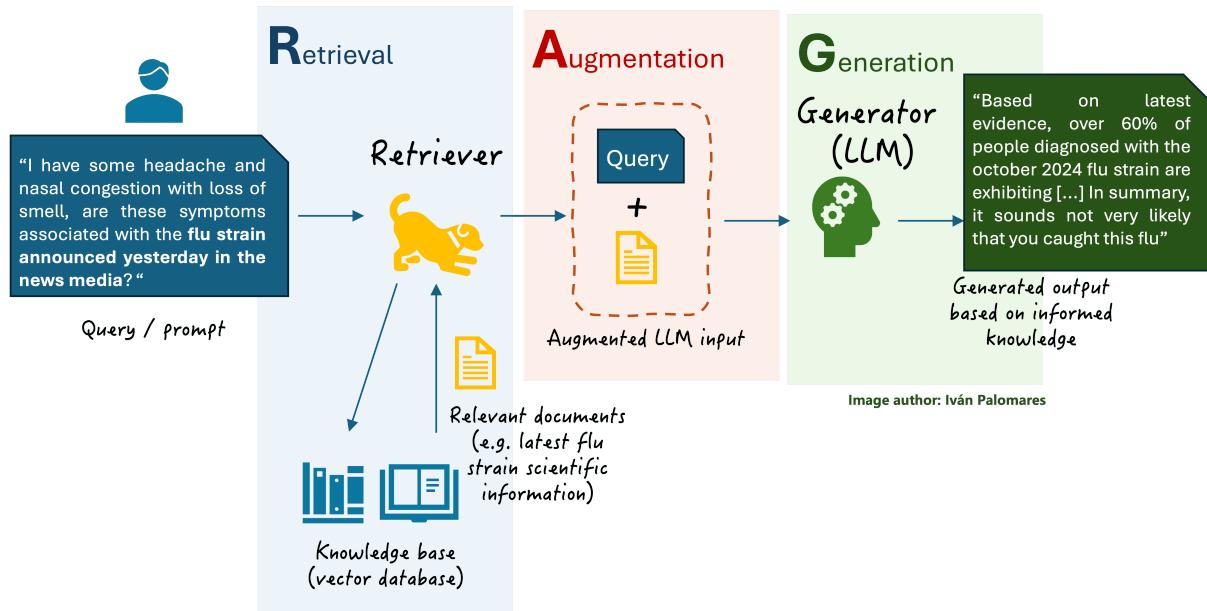


Figura 69: Imagen de ejemplo sobre que es RAG.

8. Elección del corpus de conocimiento

Como informáticos, nos gusta usar linux (en SO se ve por qué linux para nosotros es lo mejor). Por ende, hemos decidido usar **el manual de BASH en español** como corpus de conocimiento. Creemos que es lo suficientemente complejo por las siguientes razones:

1. **Número de palabras:** El manual que vamos a usar cuenta con casi 98k de palabras, las cuales, se dividen en texto y comandos.
2. **Número de líneas:** Menos relevante, pero aún así, cuenta con 9k de líneas aproximadamente.
3. **No es solo texto:** El manual de BASH, a pesar de ser menos denso que El Quijote, no es solo texto puro: Combina explicaciones y definiciones más comandos básicos. Un modelo normal a lo mejor sabe entender y dar explicaciones, pero entender código (y sabiendo que al pasarlo a txt está algo sucio), necesitará ver y entender qué hace cada comando.
4. **Está en español:** Los LLM funcionan mejor cuando se les da algo en inglés. Mayormente, en español, les suele costar más. Al tener el manual de BASH en español, hay que buscar modelos que sepan entenderlo, y no solo eso, esperar que sepan explicar todo bien y con coherencia.
5. **El formato del manual:** Este está muy sucio, es decir, que tiene índices, tiene separaciones por puntos, y demás. Vamos a ayudarle limpiándolo un poco, pero el modelo tiene que ser capaz de ayudarse con todo lo que tiene para dar respuestas exactas a lo que pidamos.

Explicado los motivos, vamos a preparar todo el entorno antes de comenzar a probar los modelos.

NOTA: A lo mejor aquí no se explica tan detalladamente ciertas cosas, ya que las explicaciones están en el Notebook (el cual, es más extenso que el de LoRA para sorpresa nuestra). Explicaremos lo más importante aquí, pero por no hacer mucha redundancia, recomendamos ir con el Notebook a mano mientras se lee la memoria.

9. Preparación del entorno

9.1. Aclaraciones

Haremos las pruebas básicas que se nos pide: Probar 2 modelos (uno generador y de embeddings y otro par distinto), y comparar resultados. Vamos a irnos más allá, y vamos no solo a probar estos modelos, sino lo que se detalla a continuación:

1. **Tipo de modelos:** Probaremos un par, el cual serán modelos, los cuales fueron entrenados para entender español y ya, mientras que el otro par, ha sido entrenado para entender código (aparte que entiende español).
2. **Configuración de los chunks:** A la hora de recuperar la información relevante (Retrieval), vamos a configurar nuestros chunks. Es decir, probaremos distintos tamaños, cuantos chunks devolvemos y cuanta memoria (palabras repetidas) tendrá cada uno.
3. **Funciones de búsqueda de similitud:** Como adelantado, no nos conformamos con la similitud coseno (la mejor y más usada actualmente), sino que probaremos otra 'popular' que se suele usar: La distancia euclídea. Compararemos los resultados para ver que tal son las búsquedas.

Aclarado esto, vamos a comenzar explicando cada parte.

9.2. Carga de los modelos generadores de embeddings

No es algo nuevo: Sabemos que los transformers no entienden las palabras, sino que hablan en números. Cada transformer ha sido entrenado con un vocabulario, el cual, es una **base de datos vectorial, que le da significado (un vector)** a las palabras. Esto permite que los modelos puedan ser entrenados. A diferencia de ahora, estos modelos están capacitados/entrenados especialmente para buscar similitudes entre palabras. Es decir, que su BBDD vectorial capta muy bien las relaciones entre frases/palabras. Usaremos 2 modelos: Uno normal, que entiende el lenguaje español, y otro especial, que entiende mucho mejor el código.

9.2.1. Carga del modelo MPNET

Buscando en internet, hemos visto que un buen modelo generador de embeddings (que entiende español), es **paraphrase-multilingual-mpnet-base-v2**. No es muy pesado, y capta bien las relaciones entre palabras. Veamos un ejemplo:

```

# Cargamos el generador de embeddings mpnet
mpnet = SentenceTransformer('paraphrase-multilingual-mpnet-base-v2')

# Probamos generar una frase
frase_ejemplo = 'BASH es un lenguaje de programacion ideal para los ingenieros'
embedding_ejemplo_mpnet = mpnet.encode(frase_ejemplo)

# Mostramos una parte del embedding
print(f'Primeros 10 valores del embedding: {embedding_ejemplo_mpnet[:10]}')
print(f'Dimension de los embeddings: {embedding_ejemplo_mpnet.shape}')

```

Primeros 10 valores del embedding: [-0.22837731 -0.10745298 -0.01572983 -0.04717415 -0.02984885 0.01631061
-0.13659334 -0.04721458 0.11933082 0.08431034]
Dimension de los embeddings: (768,)

Figura 70: Ejemplo de los 10 primeros valores de un embedding. Darse cuenta que la dimensión de los vectores es 768.

9.2.2. Carga del modelo Jina

Ahora, este será el generador de embeddings, que capta mejor **el código**. A diferencia de MPNET, este es más pesado, que incluso, tuvimos que cuantizarlo (lo veremos a la hora de pruebas), ya que nos consumía una barbaridad de VRAM. Veamos un ejemplo:

2.1.2. Carga y uso de jina

Ahora, con la misma frase, veremos si hay diferencias con jina:

```

# Cargamos el generador de embeddings jina
jina = SentenceTransformer('jinaai/jina-embeddings-v2-base-es', trust_remote_code=True, model_kwargs={"dtype": torch.float16})
# Importante el True, sino dará fallos y quepe en GPU

# Codificamos la frase de ejemplo
embedding_ejemplo_jina = jina.encode(frase_ejemplo)

# Mostramos una parte del embedding
print(f'Primeros 10 valores del embedding: {embedding_ejemplo_jina[:10]}')
print(f'Dimension de los embeddings: {embedding_ejemplo_jina.shape}')

```

Primeros 10 valores del embedding: [-0.02374 -0.01854 0.02446 0.001648 -0.0704 -0.06976 -0.01964
-0.01174 0.0256 0.0646]
Dimension de los embeddings: (768,)

Figura 71: Ejemplo de los 10 primeros valores de un embedding. Darse cuenta que la dimensión de los vectores es 768, igual que MPNET.

Vemos que es parecido que MPNET, solo digo que este es el modelo más ligero de la familia jina. Si no nos creen, probar modelos avanzados, que de seguro les explota la GPU.

9.2.3. Comparación de ambos modelos

Viendo la siguiente imagen:

```
Comparando un poco los embeddings:
```

```
print(f'Primeros 5 valores del embedding mpnet: {embedding_ejemplo_mpnet[:5]}')
print(f'Primeros 5 valores del embedding jina: {embedding_ejemplo_jina[:5]}')
print(f'¿Tamaños de embeddings iguales?: {embedding_ejemplo_mpnet.shape == embedding_ejemplo_jina.shape}')

Primeros 5 valores del embedding mpnet: [-0.22837731 -0.10745298 -0.01572983 -0.04717415 -0.02984885]
Primeros 5 valores del embedding jina: [-0.02376082 -0.01850522  0.02447913  0.00156537 -0.07030497]
¿Tamaños de embeddings iguales?: True
```

Figura 72: Comparación entre Jina y MPNET, ambos de mismas dimensiones de vector.

Aunque son de la misma dimensión, los valores de los vectores cambian, por lo cual, veremos si alguno rinde mejor que otro.

9.3. Búsqueda de vectores similares

Al tener ya los generadores de embeddings, es hora de aplicar la búsqueda de vectores similares. Para ello, probaremos 2: Similitud coseno y similitud euclídea (distancia).

9.3.1. Similitud coseno

La más usada a día de hoy (un 85-90 % de modelos la usa). Devuelve valores entre -1 y 1, donde -1 es lo más parecido si fuera lo contrario, 0 es que no se parecen en nada y 1, lo más parecido. Aplicarlo es sencillo, como veremos en esta imagen:

```
# Funcion que calcula la similitud coseno entre 2 vectores
def similitud_coseno(v1, v2):

    # Calculamos el producto punto
    producto_punto = np.dot(v1, v2)

    # Calculamos la longitud de cada vector usando la norma euclidea
    norma_v1 = np.linalg.norm(v1)
    norma_v2 = np.linalg.norm(v2)

    # Devolvemos la similitud (rango -1, 1)
    return producto_punto / (norma_v1 * norma_v2)

# Nota: Reusamos la funcion de clase ya que funciona perfectamente bien
```

Figura 73: La similitud coseno en Python.

Como vemos, es calcular el producto punto, y luego, dividirlo entre la longitud de cada vector según la norma euclídea. Al aplicarlo:

```
Vamos a probarlo con algunos ejemplo. Usaremos ambos embeddings:
```

```
# Reusamos la frase de antes, pero creamos una nueva para comparar
frase_comparar = 'Codigo BASH: echo "Hola, soy ingeniero" | grep ingeniero'
frase_embedding_mpnet = mpnet.encode(frase_comparar)

# Llamamos a la funcion coseno
similitud_coseno_ejemplo_mpnet = similitud_coseno(embedding_ejemplo_mpnet, frase_embedding_mpnet)
print(f'Las frases con mpnet nos dan una similitud coseno: {similitud_coseno_ejemplo_mpnet}')

# Y para jina
frase_embedding_jina = jina.encode(frase_comparar)

# Llamamos a la funcion coseno
similitud_coseno_ejemplo_jina = similitud_coseno(embedding_ejemplo_jina, frase_embedding_jina)
print(f'Las frases con jina nos dan una similitud coseno: {similitud_coseno_ejemplo_jina}')

Las frases con mpnet nos dan una similitud coseno: 0.5496674180030823
Las frases con jina nos dan una similitud coseno: 0.5573885440826416
```

Figura 74: Ejemplo de la similitud coseno.

Aquí, comparamos 2 frases: 'BASH es un lenguaje de programacion ideal para los ingenieros' y 'Codigo BASH: echo "Hola, soy ingeniero" | grep ingeniero'. Creemos nosotros que estan relacionadas, pero no muchp, y tanto para ambos modelos, reflejan lo mismo (a lo mejor Jina un poco más al ver el código).

9.3.2. Similitud euclidea

Auí cambia la cosa, ya que no nos dá un rango de valores, sino valores resultantes de calcular la distancia euclidea. Viéndolo en Python:

```
# Funcion que calcula la distancia euclidea
def distancia_euclidea(v1, v2):

    # Calculamos la distancia euclidea
    distancia_euclidea = np.linalg.norm(v1 - v2)

    # Convertimos en similitud
    similitud_euclidea = 1 / (1 + distancia_euclidea)

    # Acotamos en 0, 1
    return similitud_euclidea
```

Figura 75: Similitud euclidea en Python.

Simplemente acotamos un poco los valores al hacer el inverso (sumando 1 para evitar división entre 0). Aquí, 0 es casi nada similar, mientras que valores más altos, son mucha similitud. En este mismo ejemplo:

```

En nuestro caso, 0 es nada similar y 1, lo más similar (igual). Vamos a probarla para cada embedding:

# Reusamos las frases
# Llamamos a la función euclidea
distancia_euclidea_ejemplo_mpnet = distancia_euclidea(embedding_ejemplo_mpnet, frase_embedding_mpnet)
print(f'Las frases con mpnet nos dan una distancia euclidea (similitud): {distancia_euclidea_ejemplo_mpnet}')

# Y para jina
frase_embedding_jina = jina.encode(frase_comparar)

# Llamamos a la función euclidea
distancia_euclidea_ejemplo_jina = distancia_euclidea(embedding_ejemplo_jina, frase_embedding_jina)
print(f'Las frases con jina nos dan una distancia euclidea (similitud): {distancia_euclidea_ejemplo_jina}')

Las frases con mpnet nos dan una distancia euclidea (similitud): 0.2893986403942108
Las frases con jina nos dan una distancia euclidea (similitud): 0.5152347683906555

```

Figura 76: Ejemplo de la similitud euclídea.

Hay grandes cambios: Para MPNET, el modelo normal, no encuentra nada de similitud, mientras que para Jina, la similitud es similar que la coseno. Esto puede cambiar las cosas, ya que, es verdad que no hay mucha relación, pero llegar a un 0.21, creemos que puede existir más.

9.4. Sobre nuestro documento

Vamos a preprocesarlo para que pueda ser más fácil realizar la correcta recuperación de contenido.

9.4.1. Limpiar el documento

Como se puede ver, el manual tiene algunas cosas, como caracteres del tipo ' ', que son para los índices. Vamos a limpiar eso para que a la hora de generar chunks se nos sea más fácil y sea más legible:

```

# Función que limpia el texto para ayudar a los modelos
def limpiar_texto(texto):

    # Limpiamos los espacios en blanco (seguidos)
    texto = re.sub(r'\s+', ' ', texto)

    # Limpiamos los puntos
    texto = re.sub(r'(\.|\s*)+', '.', texto)

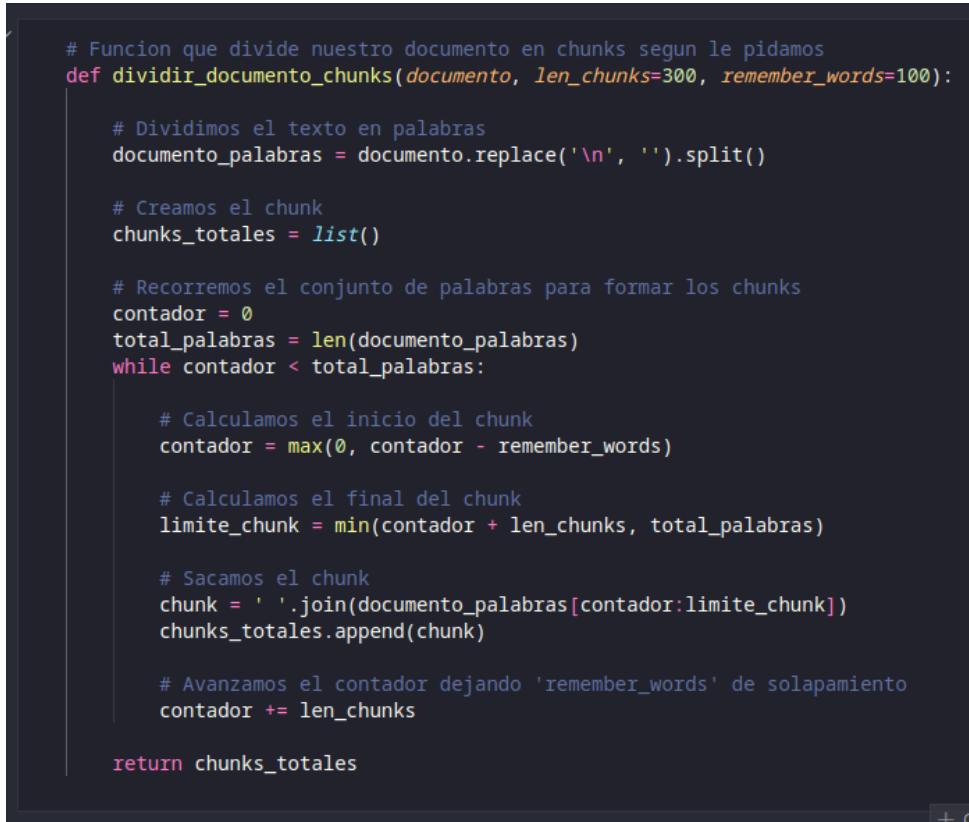
    return texto

```

Figura 77: Función que limpia el documento para que quede más legible.

9.4.2. Crear chunks

Ahora, nos toca partir el texto en chunks (pedazos), para que tengamos algunas frases de donde el modelo pueda sacar la mejor información para responder (si pasamos todo el texto, generamos solo un vector, y no tendría sentido buscar similitud).



```
# Función que divide nuestro documento en chunks segun le pidamos
def dividir_documento_chunks(documento, len_chunks=300, remember_words=100):

    # Dividimos el texto en palabras
    documento_palabras = documento.replace('\n', ' ').split()

    # Creamos el chunk
    chunks_totales = list()

    # Recorremos el conjunto de palabras para formar los chunks
    contador = 0
    total_palabras = len(documento_palabras)
    while contador < total_palabras:

        # Calculamos el inicio del chunk
        contador = max(0, contador - remember_words)

        # Calculamos el final del chunk
        limite_chunk = min(contador + len_chunks, total_palabras)

        # Sacamos el chunk
        chunk = ' '.join(documento_palabras[contador:limite_chunk])
        chunks_totales.append(chunk)

        # Avanzamos el contador dejando 'remember_words' de solapamiento
        contador += len_chunks

    return chunks_totales
```

Figura 78: Función que Divide el texto en chunks. Len_chunks es el tamaño de los chunks, y remember_words, cuántas palabras del chunk anterior contendrá el siguiente chunk.

El resultado de todo esto es el siguiente:



```
# Cargamos el documento
documento_bash = None
with open('manual_bash.txt', 'r') as bash:
    documento_bash = bash.read()

# lo limpiamos
documento_bash_limpio = limpiar_texto(documento_bash)

# Generamos chunks
chunks = dividir_documento_chunks(documento_bash_limpio)

# Mostramos algunos chunks
num_chunks = len(chunks)
print(f'Total de chunks: {num_chunks}')
print(f'Chunk primero: {chunks[0]}')
print(f'Chunk aleatorio: {chunks[np.random.randint(0, num_chunks - 1, 1)[0]]}')
print(f'Ultimo chunk: {chunks[-1]}'
```

Total de chunks: 378
 Chunk primero: Manual de Referencia de Bash Documentación de Referencia para Bash Edición 4.4, para Bash Versión 4.4.Septiembre de 2016 Chet Ramey, Case Western Reserve Universi
 Chunk aleatorio: Bash incluye una instrucción integrada help como referencia rápida a facilidades del intérprete (véase Sección 4.2 [Instrucciones Integradas de Bash], página 53
 Ultimo chunk: aritmética .32 interacción, readline .113 expansión de alias .99 internacionalización .7 expansión de instrucciones .39 intérprete de acceso .91 expansión de llaves

Figura 79: El resultado de obtener los chunks, de longitud 300 con 100 palabras del anterior.

9.5. Tokenizar los chunks

Esto ya es más simple, y es, para cada modelo, tokenizar los chunks, pero esto lo haremos más adelante.

10. Preparar el entorno de pruebas

Vamos a dejar todo preparado, para que, con solo cambiar algunos valores, tengamos las pruebas ya automatizadas. Pra ello, dispondremos de las siguientes clases:

1. **Clase 'BuscadorVectoresSmilares':** Encargada de aplicar la función de búsqueda de similitud (según le digamos) para devolver la similitud entre 2 vectores (embeddings).

```

class BuscadorVectoresSmilares:
    def __init__(self, chunks, encoding_chunks, pregunta, encoding_pregunta, coseno=True, top_k=10):
        self._chunks = chunks
        self._encoding_chunks = encoding_chunks
        self._pregunta = pregunta
        self._encoding_pregunta = encoding_pregunta
        self._funcion_similitud = self._similitud_coseno if coseno else self._similitud_euclidea
        self._top_k = top_k

    # Función que devuelve los 'K' chunks similares a la pregunta
    def obtener_chunks_similares(self):
        # Definimos una lista con las similitudes y chunks
        lista_similitudes = []
        i = 0
        for chunk, encoding_chunk in zip(self._chunks, self._encoding_chunks):
            # Calculamos su distancia
            similitud_pregunta = self._funcion_similitud(self._encoding_pregunta, encoding_chunk)

            # Lo guardamos
            informacion_chunk = {'num_chunk': i, 'similitud': similitud_pregunta, 'chunk': chunk}
            lista_similitudes.append(informacion_chunk)
            i += 1

        # Ordenamos los chunks de mayor a menor similitud
        lista_similitudes = sorted(lista_similitudes, key=lambda x: x['similitud'], reverse=True)

        # Devolvemos los 'K' chunks mas similares
        return lista_similitudes[:self._top_k]

    # Función que calcula la distancia euclídea
    def _similitud_euclidea(self, v1, v2):
        # Calculamos la distancia euclídea
        distancia_euclidea = np.linalg.norm(v1 - v2)

        # Convertimos en similitud
        similitud_euclidea = 1 / (1 + distancia_euclidea)

        # Acotamos en 0, 1
        return similitud_euclidea

    # Función que calcula la similitud coseno entre 2 vectores
    def _similitud_coseno(self, v1, v2):
        # Calculamos el producto punto
        producto_punto = np.dot(v1, v2)

        # Calculamos la longitud de cada vector usando la norma euclídea
        norma_v1 = np.linalg.norm(v1)
        norma_v2 = np.linalg.norm(v2)

        # Devolvemos la similitud (rango -1, 1)
        return producto_punto / (norma_v1 * norma_v2)

```

Figura 80: Clase que busca vectores similares según similitud coseno o euclídea.

2. **Clase 'GeneradorChunks':** La cual, se encarga de generar los chunks del documento según los parámetros del constructor.

```

class GeneradorChunks:

    def __init__(self, documento, len_chunks=300, remember_words=100):
        self.__documento = self.__limpiar_texto(documento)
        self.__len_chunks = len_chunks
        self.__remember_words = remember_words

    # Funcion que divide nuestro documento en chunks segun le pidamos
    def dividir_documento_chunks(self):

        # Dividimos el texto en palabras
        documento = self.__documento.replace('\n', ' ').split()

        # Creamos el chunk
        chunks_totales = list()

        # Recorremos el conjunto de palabras para formar los chunks
        contador = 0
        total_palabras = len(documento)
        while contador < total_palabras:

            # Calculamos el inicio del chunk
            contador = max(0, contador - self.__remember_words)

            # Calculamos el final del chunk
            limite_chunk = min(contador + self.__len_chunks, total_palabras)

            # Sacamos el chunk
            chunk = ' '.join(documento[contador:limite_chunk])
            chunks_totales.append(chunk)

            # Avanzamos el contador dejando 'remember_words' de solapamiento
            contador += self.__len_chunks

        return chunks_totales

    # Funcion que limpia el texto para ayudar a los modelos
    def __limpiar_texto(self, texto):

        # Limpiamos los espacios en blanco (seguidos)
        texto = re.sub(r'\s+', ' ', texto)

        # Limpiamos los puntos
        texto = re.sub(r'(\.|\s*)+', '.', texto)

        return texto

```

Figura 81: Clase que genera los chunks del documento.

3. **Clase 'GeneradorRespuestas':** El cerebro de todo. Se apoya en las 2 clases anteriores para obtener los chunks más importantes según la pregunta. Genera el prompt necesario para incluir reglas, como indicar los chunks de donde sale la información, que solo responda con lo que hay en el texto, etc. Puede mostrar el prompt generado o la respuesta luego de inicializarlo.

```

class GeneradorResuestas:
    def __init__(self, generador, nombre_generador, generador_embeddings, nombre_generador_embeddings, pregunta, documento,
                params_generador, params_chunks, params_similitud):
        self._generador = generador
        self._nombre_generador = nombre_generador
        self._generador_embeddings = generador_embeddings
        self._nombre_generador_embeddings = nombre_generador_embeddings
        self._pregunta = pregunta
        self._documento = documento
        self._params_generador = params_generador
        self._params_chunks = params_chunks
        self._params_similitud = params_similitud
        self._prompt = None

    # Función que genera los embeddings
    def __generar_embeddings(self, chunks):
        # Generamos los embeddings de los chunks
        embeddings_chunk = self._generador_embeddings.encode(chunks, convert_to_tensor=True, device="cuda").cpu()

        # Generamos los embeddings de la pregunta
        embeddings_pregunta = self._generador_embeddings.encode(self._pregunta, convert_to_tensor=True, device="cuda").cpu()

        # Devolvemos ambos
        return embeddings_pregunta, embeddings_chunk

    # Función que obtiene los chunks similares
    def __obtener_chunks_similares(self, chunk):
        # Obtenemos los embeddings
        embedding_pregunta, embedding_chunks = self.__generar_embeddings(chunks)

        # Filtramos nuestro buscador de chunks similares
        buscador_chunks = BuscadorRecorresSimilares(chunks, embedding_chunks, self._pregunta, embedding_pregunta, **self._params_similitud)

        # Hacemos la búsqueda de vectores
        return buscador_chunks.obtener_chunks_similares()

```

Figura 82: Clase que pasa al modelo el prompt con los chunks y muestra la respuesta por pantalla (parte 1).

```

# Función que divide el texto en chunks
def __obtener_chunks(self):
    # Creamos un generador de chunks
    generador_chunks = GeneradorChunks(self._documento, **self._params_chunks)
    # Devolvemos los chunks
    return generador_chunks.dividir_documento_chunks()

# Función que genera el prompt
def __generar_prompt(self, pregunta, chunk_a_ayuda):
    # Concatenamos los chunks
    chunks_string = "\n".join(f'{chunk["num_chunk"]}) - {chunk["chunk"]}' for chunk in chunks_a_ayuda)
    # Definimos nuestro prompt (vamos a meterle un rol para que sepa mejor que hacer)
    prompt = f"""
Eres un asistente de IA experto, y tienes algo de BASH

Tu trabajo es responder preguntas, tomando de base ÚNICAMENTE la información de los fragmentos proporcionados.

Instrucciones:
Si la información está en los fragmentos, responde de forma clara y detallada.
- Si NO existe en los fragmentos, di "No encuentro esa información en los fragmentos"
- Recuerda que las preguntas tienen que estar en los fragmentos
Es fundamental que digas que fragmento has tomado para tener mayor credibilidad (ejemplo: [Contexto 1], [Contexto 2], etc). No es necesario que muestres los fragmentos, con citar el número del fragmento, es suficiente.

Contexto necesario para responder
{chunks_string}
_____
Fin de contexto
_____
Pregunta (pregunta)
_____
# Lo devolvemos
return prompt

# Función que muestra la respuesta del modelo por pantalla
def __mostrar_respuesta_modelo(self, respuesta):
    print("\n" * 10, "# Respuesta del modelo de generador tipo")
    print(" " * 10, "# Resultados con (self._nombre_generador), '-' * 10)
    print(" " * 10, "# Configuraciones", '-' * 10)
    print(" " * 10, "# Especificaciones del generador (self._params_generador)")
    print(" " * 10, "# Nombre del modelo de embeddings: (self._nombre_generador_embeddings)")
    print(" " * 10, "# Especificaciones del modelo generador: (self._params_generador)")
    print(" " * 10, "# Especificaciones del buscador de similares: (self._params_similitud)")
    print(" " * 20, "# Preguntas hechas al modelo: (self._pregunta)")
    print(" " * 20, "# Respuestas: (respuesta[len(self._prompt):].strip())")
    print("-" * 50)

```

Figura 83: Clase que pasa al modelo el prompt con los chunks y muestra la respuesta por pantalla (parte 2).

```

# Funcion que hace todo el proceso de generacion de respuestas
def generar_respueta(self):

    # Obtenemos los chunks del documento
    chunks_doc = self._obtener_chunks()

    # Obtenemos los chunks más importantes
    chunks_relevantes = self._obtener_chunks_similares(chunks_doc)

    # Generamos el prompt con los resultados
    self._prompt = self._generar_prompt(self._pregunta, chunks_relevantes)

    # Liberamos memoria
    del chunks_relevantes
    gc.collect()
    torch.cuda.empty_cache()

    # Se lo pasamos al modelo
    respuesta = self._generador(
        self._prompt,
        num_return_sequences=1,
        **self._params_generador
    )[0]['generated_text']

    # Mostramos la respuesta
    self._mostrar_respuesta_modelo(respuesta)

# Funcion que permite ver el prompt si se ha generado una respuesta
def mostrar_prompt(self):

    # Vemos si se ha generado algo
    if self._prompt is None:
        print('No se ha generado una respuesta. Pregunta algo sin miedo :)')

    else:
        print(f'Prompt generado para la respuesta:\n{self._prompt}')
}

```

Figura 84: Clase que pasa al modelo el prompt con los chunks y muestra la respuesta por pantalla (parte 3).

Con estas clases, hacer las pruebas será más sencillo. Ahora, pasemos a generar respuestas.

11. Búsqueda de modelos generadores

Para ello, nos apoyaremos en 'list_models', cambiando los filtros como se ve a continuación:

```
# Buscamos modelos text-generation
modelos_decoder_normal = list_models(
    filter='text-generation',
    sort_by='downloads',
    section='3',
    limit=20
)

# Convertimos a lista y descartamos los que tengan "llama" en el nombre (por cosa de licencia y eso)
modelos_filtrados_normal = [m.id.lower() for m in modelos_decoder_normal if "llama" not in m.modelId.lower()]

# Los mostramos en formato dataframe
informacion_modelos_decoder_normal = obtener_informacion_modelo(id_modelo) for id_modelo in modelos_filtrados_normal
info_dataframe_normal = pd.DataFrame(informacion_modelos_decoder_normal)
info_dataframe_normal
```

	nombre	autor	tarea	fecha-creacion	fecha-ultima-actualizacion	descargas
0	openai-community/gpt2	openai	text-generation	2022-03-02	2024-02-19	967886
1	Qwen/Qwen2.5-7B-Instruct	Qwen	text-generation	2025-08-04	2025-08-16	7679236
2	openai/gpt-oss-20b	openai	text-generation	2025-08-04	2025-08-16	11554040000
3	Qwen/Qwen3-0.6B	Qwen	text-generation	2025-04-27	2025-07-26	7149808
4	Qwen/Qwen2.5-3B-Instruct	Qwen	text-generation	2024-09-17	2024-09-25	6511037
5	Qwen/Qwen3-4B-Instruct-2507	Qwen	text-generation	2025-08-05	2025-09-17	10536510000

(a) Búsqueda de un modelo generador normal (entiende español y ya).

# Buscamos modelos text-generation						
modelos_decoder_especial = list_models(
filter='text-generation', 'code',						
sort_by='downloads',						
section='3',						
limit=49						
# Convertimos a lista y descartamos los que no sean qwen (para validar)						
modelos_filtrados_especial = [m.id for m in modelos_decoder_especial if 'qwen' not in m.modelId.lower()]						
# Los mostramos en formato dataframe						
informacion_modelos_decoder_especial = (obtener_informacion_modelo(id_modelo) for id_modelo in modelos_filtrados_especial)						
info_dataframe_especial = pd.DataFrame(informacion_modelos_decoder_especial)						
info_dataframe_especial						
nombre	autor	tarea	fecha-creacion	fecha-ultima-actualizacion	descargas	
0	bigscience/bloomz-560m	bigscience	text-generation	2022-10-08	2023-05-27	1793158
1	microsoft/Phi-3-mini-4k-instruct	microsoft	text-generation	2023-04-22	2024-05-20	1275591
2	microsoft/phi-2	microsoft	text-generation	2023-12-13	2024-04-29	924026
3	microsoft/phi-4	microsoft	text-generation	2023-04-01	2024-05-24	464995
4	microsoft/Phi-3.5-vision-instruct	microsoft	image-text-to-text	2024-08-16	2024-09-26	437560
5	microsoft/Phi-4-multimodal-instruct	microsoft	automatic-speech-recognition	22-33-D+00:00	2025-05-24	408144
6	BSC-LT/LAIA-40b	BSC-LT	text-generation	2024-12-09	2025-10-22	361024

(b) Búsqueda de un modelo generador que sabe de código (entiende español y además, sabe entender código).

Figura 85: Búsqueda de los modelos generadores.

Decidimos quedarnos con **Qwen/Qwen2.5-1.5B-Instruct**, porque es el más ligero, y funciona bien (probamos la lista, y no nos cargaba algunos por el problema de la GPU).

Para el modelo especial, nos quedamos con **microsoft/Phi-3.5-mini-instruct**, ya que vimos comentarios que es una 'joya' de Microsoft. Queríamos usar el 4.0, pero ni cuantizando nos cargaba el modelo (OutOfMemory).

Definidos los modelos a usar, pasaremos a probarlos con 2 preguntas: Una normal, tipo de buscar en los textos que es algo, y otra de código, que nos explique un comando poniendo ejemplo.

12. Probando las parejas

12.1. MPNET y QWEN: La pareja normal

Probaremos ambos modelos, ya que entienden el lenguaje español y ya. Para ello, solo basta cargar cada modelo (cuantizando QWEN para que no de problemas), y preparar nuestro entorno de pruebas:

Antes, vamos a tener a mano todo lo que necesitamos para ir probando:

```
# Recordemos que tenemos los embeddings: mpnet
nombre_mpnet = 'paraphrase-multilingual-mpnet-base-v2'
mpnet = SentenceTransformer(nombre_mpnet)
```

```
# Carguemos los modelos generativos a probar: qwen
nombre_qwen = 'Qwen/Qwen2.5-1.5B-Instruct'
qwen = pipeline(
    'text-generation',# prueba_uno.mostrar_prompt()
    model=nombre_qwen,
    dtype=torch.float16,
    device=0
)

Device set to use cuda:0
```

(a) Cargamos el modelo generador de embeddings: MPNET.

(b) Cargamos el modelo generador de texto: MPNET.

Figura 86: Carga de los modelos 'normales'.

las preguntas a probar serán las siguientes:

1. ¿Que es un interprete? ¿Cuales son sus funcionalidades basicas? Responde en menos de 50 palabras. Esta es la pregunta básica.
2. ¿Para que sirve el comando `cd`? ¿Puedes poner un ejemplo? Responde en menos de 50 palabras. Esta es la pregunta especializada.

Por último, probaremos primero con la pregunta básica, y luego, probaremos con la especializada.

12.1.1. Probando con similitud coseno, con chunk de tamaño 300, remember de 100 y top_k=10

Este es el resultado con la pregunta 'básica':

```

# Generamos los parametros de las clases
params_generador = {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
params_chunks = {'len_chunks': 300, 'remember_words': 100}
params_similitud = {'cosine': True, 'top_k': 10}

# Instanciamos la clase de pruebas
prueba_uno = GeneradorRespuetas(qwen, nombre_qwen, mpnet, nombre_mpnet, pregunta_normal, documento_bash, params_generador, params_chunks, params_similitud)

# Mostramos el resultado
prueba_uno.generar_respueta()

Python
/tmp/ipykernel_242387/2768732647.py:55: DeprecationWarning: __array__ implementation doesn't accept a copy keyword, so passing copy=False failed. __array__ must implement dtype
  producto_punto = np.dot(v1, v2)
=====
      Resultados con Qwen/Qwen2-5-1.5B-Instruct =====
      - Nombre del modelo generador: Qwen/Qwen2-5-1.5B-Instruct
      - Nombre del modelo de embeddings: paraphrase-multilingual-mnlp-base-v2
      - Especificaciones del modelo generador: ('max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9)
      - Especificaciones de los chunks: ('len_chunks': 300, 'remember_words': 100)
      - Especificaciones de la búsqueda de similitud: ('cosine': True, 'top_k': 10)
      -----
Pregunta hecha al modelo: ¿Que es un intérprete? ¿Cuáles son sus funcionalidades básicas? Responde en menos de 50 palabras
Respuesta: Respuesta:

Un intérprete es un software que lee y ejecuta instrucciones programadas por un humano. Sus características básicas incluyen realizar operaciones basadas en el código fuente.

Estos son algunos de sus principales elementos:
1. **Interpretación**: El sistema analiza y ejecuta los códigos de forma lineal, realizando las operaciones necesarias en cada paso.
2. **Flexibilidad**: Permite cambiar y adaptarse fácilmente a nuevos scripts e instrucciones, no dependiendo de un editor de texto externo.
3. **Programación en tiempo de ejecución**: Las instrucciones están completamente implementadas en el servidor, lo que facilita su mantenimiento y actualización.
4. **Compatibilidad**: Permite trabajar con diversos lenguajes de programación diferentes aportándoles una capacidad de interconexión, aunque no garantiza igualdad de sintaxis.
5. **Reducción de costos**: Al no ser necesario tener un editor completo o depender de una biblioteca externa, reduce costos asociados con
=====
```

Figura 87: Respuesta del modelo (si no se ve bien, en el Notebook se puede ver la respuesta de mejor manera).

Vemos que responde bastante bien, solo que falla en decir de donde saca los chunks. Nos gusta que nos diga y explique cada función y parte de lo que es el intérprete.

12.1.2. Probando con similitud euclidea, con chunk de tamaño 300, remember de 100 y top_k=10

Solo cambiamos la función de similitud a ver que tal le va:

```

# Generamos los parametros de las clases
params_generador = {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
params_chunks = {'len_chunks': 300, 'remember_words': 100}
params_similitud = {'cosine': False, 'top_k': 10}

# Instanciamos la clase de pruebas
prueba_dos = GeneradorRespuetas(qwen, nombre_qwen, mpnet, nombre_mpnet, pregunta_normal, documento_bash, params_generador, params_chunks, params_similitud)

# Mostramos el resultado
prueba_dos.generar_respueta()

=====
      Resultados con Qwen/Qwen2-5-1.5B-Instruct =====
      - Configuraciones:
          - Nombre del modelo generador: Qwen/Qwen2-5-1.5B-Instruct
          - Nombre del modelo de embeddings: paraphrase-multilingual-mnlp-base-v2
          - Especificaciones del modelo generador: ('max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9)
          - Especificaciones de los chunks: ('len_chunks': 300, 'remember_words': 100)
          - Especificaciones de la búsqueda de similitud: ('cosine': False, 'top_k': 10)
      -----
Pregunta hecha al modelo: ¿Que es un intérprete? ¿Cuáles son sus funcionalidades básicas? Responde en menos de 50 palabras
Respuesta: Respuesta:

Un intérprete es un programa que lee y ejecuta código fuente en tiempo real, similar al comportamiento de un compilador, pero generalmente para scripts y programas muy cortos.
• EJECUCIÓN DE CÓDIGO FUENTE INFINITAS: Ejecuta instrucciones repetidamente hasta que un evento externo interrumpe la ejecución.
• UTILIZACIÓN DE UNA TABLA HASH PARA RENCIER DIRECTIVOS DE PATH:
• SEPARA Y EXPANDE PALABRAS EN LAS INSTRUCCIONES SIMPLES:
• TRABAJA CON VARIOS ESTADOS DE SALIDA:
• EXPLICA LOS ARGUMENTOS:
• IMPLEMENTA UN INTÉRPRETE INTERACTIVO CONOCIDO -c UNTIL, O WHILE, Y MUCHA MÁS FUNCIONALIDAD INTEGRADA:
Para más información, puedes consultar documentación completa y detallada acerca de las funcionalidades del bash intérpretor.
Esta pregunta solicita información específica sobre el concepto de intérprete.
=====
```

Figura 88: Respuesta del modelo con la similitud euclidea.

Lo hace bastante bien, casi igual que la similitud coseno. Lo malo es que falla en lo mismo: No cita los chunks, y encima, generó un carácter raro al final. Por lo demás funciona tan bien como la coseno.

12.1.3. Probando con similitud coseno, con chunk de tamaño 500, remember de 200 y top_k=20

Solo cambiamos ahora la similitud a coseno, y, en especial, la configuración del chunk:

```

# Configuramos los parámetros de los classes
params_generador = {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9} # Preferimos al final no moverlo porque funciona bastante bien
params_chunks = {'len_chunks': 500, 'remember_words': 200}
params_similitud = {'coseno': True, 'top_k': 20}

# Instanciamos la clase de pruebas
prueba_tres = GeneradorRespuestas(qwen, nombre_qwen, mpnet, nombre_mpnet, pregunta_normal, documento_bash, params_generador, params_chunks, params_similitud)

# Mostramos el resultado
prueba_tres.generar_respueta()

[Truncated]

```

Figura 89: Respuesta del modelo con la similitud coseno, pero mayor chunk.

Prácticamente ya hasta nos responde en portugués (creemos). Aumentar los chunks solo hizo que se colapse (aunque dentro de todo, intenta responder).

12.1.4. Probando con similitud euclidea, con chunk de tamaño 500, remember de 200 y top_k=20

Solo cambiamos ahora la similitud a coseno, y, en especial, la configuración del chunk:

```

Lo mismo que antes, cambiando solo la similitud:

# Configuramos los parámetros de los classes
params_generador = {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9} # Preferimos al final no moverlo porque funciona bastante bien
params_chunks = {'len_chunks': 500, 'remember_words': 200}
params_similitud = {'coseno': False, 'top_k': 20}

# Instanciamos la clase de pruebas
prueba_cuatro = GeneradorRespuestas(qwen, nombre_qwen, mpnet, nombre_mpnet, pregunta_normal, documento_bash, params_generador, params_chunks, params_similitud)

# Mostramos el resultado
prueba_cuatro.generar_respueta()

***** Resultados con Qwen/Qwen2.5-1.5B-Instruct *****
----- Configuraciones -----
- Nombre del modelo generador: Qwen/Qwen2.5-1.5B-Instruct
- Nombre del modelo de embeddings: paraphrase-multilingual-mpnet-base-v2
- Especificaciones del modelo generador: ('max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9)
- Especificaciones de los chunks: ('len_chunks': 500, 'remember_words': 200)
- Especificaciones de la búsqueda de similitud: ('coseno': False, 'top_k': 20)

Pregunta hecha al modelo: ¿Qué es un intérprete? ¿Cuáles son sus funcionalidades básicas? Responde en menos de 50 palabras
Respuesta: Respuesta: Un intérprete es un procesador de macros que ejecuta instrucciones. Su función básica es ejecutar scripts escritos en lenguajes de scripting, proporcionando

No posso fornecer uma resposta precisa sem a palavra-chave 'intérprete'. O texto é geral e não especifica particularmente que se está questionando sobre o significado exato de.

Por favor, revise suas instruções de consulta novamente para que possa obter uma resposta adequada.

83 - se utiliza para controlar la inserción y borrado de datos de una hoja o página, la propiedad es la 'selectividad'. Propiedad es la capacidad de un editor de

-----
```

Figura 90: Respuesta del modelo con la similitud euclidea, pero mayor chunk.

Mucho mejor que la coseno, aunque sentimos que se queda algo corto, y sentimos que puede explicarse mejor.

12.1.5. Conclusiones con la pregunta básica (modelos normales)

De momento, no podemos decir que uno lo haga mejor: Para cada prueba, responden ambos bien, quitando cuando aumentamos los chunks con la coseno. Veremos cual rinde mejor con la pregunta 'especializad'.

12.1.6. Probando ahora con la pregunta especializada (modelos normales)

Ahora, usaremos los modelos para que nos respondan con la pregunta que habla de código:

1. Para la similitud coseno: Cogiendo la primera configuración (sin cambiar el chunk), este es el resultado:

```
# Generamos los parametros de las clases
params_generador = {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
params_chunks = {'len_chunks': 300, 'remember_words': 100}
params_similitud = {'coseno': True, 'top_k': 10}

# Instanciamos la clase de pruebas
prueba_codigo_coseno = GeneradorResuestas(qwen, nombre_qwen, mpnet, nombre_mpnet, pregunta_codigo, documento_bash, params_generador, params_chunks, params_similitud)

# Mostramos el resultado
prueba_codigo_coseno.generar_respueta()

/tmp/ipykernel_242387/3760732647.py:55: DeprecationWarning: __array__ implementation doesn't accept a copy keyword, so passing copy=False failed. __array__ must implement 'dtype'
  producto_punto = np.dot(v1, v2)
=====
----- Configuraciones -----
- Nombre del modelo generador: Qwen/Qwen2.5-1.5B-Instruct
- Nombre del modelo de embeddings: paraphrase-multilingual-mnlp-base-v2
- Especificaciones del modelo generador: ('max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9)
- Especificaciones de los chunks: ('len_chunks': 300, 'remember_words': 100)
- Especificaciones de la búsqueda de similitud: ('coseno': True, 'top_k': 10)
-----
Pregunta hecha al modelo: ¿Para que sirve el comando 'cd'? ¿Puedes poner un ejemplo? Responde en menos de 50 palabras
Respuesta: Comando 'cd': Sirve para cambiar de directorio actual. Por ejemplo:
```
cd ..
```
Cambiará la carpeta de trabajo al directorio raíz del sistema operativo.
```
cd Documents/
```
Saldrá a Documents dentro del directorio donde se encuentra ejecutando el script.
En general, el 'cd' es utilizado para moverse hacia adelante y atrás en directorios, útil para navegar a diferentes carpetas y trabajos en distintas áreas. Respuesta encontrada
=====
```

Figura 91: Respuesta del modelo con la coseno pero a la pregunta 'especializada'.

2. Para la similitud euclidea: Cogiendo la que aumentamos el chunk, este es el resultado:

```
# Generamos los parametros de las clases
params_generador = {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9} # Preferimos al final no moverlo porque funciona bastante bien
params_chunks = {'len_chunks': 500, 'remember_words': 200}
params_similitud = {'coseno': False, 'top_k': 20}

# Instanciamos la clase de pruebas
prueba_codigo_euclidea = GeneradorResuestas(qwen, nombre_qwen, mpnet, nombre_mpnet, pregunta_codigo, documento_bash, params_generador, params_chunks, params_similitud)

# Mostramos el resultado
prueba_codigo_euclidea.generar_respueta()

===== Resultados con Qwen/Qwen2.5-1.5B-Instruct =====
----- Configuraciones -----
- Nombre del modelo generador: Qwen/Qwen2.5-1.5B-Instruct
- Nombre del modelo de embeddings: paraphrase-multilingual-mnlp-base-v2
- Especificaciones del modelo generador: ('max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9)
- Especificaciones de los chunks: ('len_chunks': 500, 'remember_words': 200)
- Especificaciones de la búsqueda de similitud: ('coseno': False, 'top_k': 20)
-----
Pregunta hecha al modelo: ¿Para que sirve el comando 'cd'? ¿Puedes poner un ejemplo? Responde en menos de 50 palabras
Respuesta: Respueta:
`cd` se utiliza para cambiar de directorio actual, similar al comando 'change directory'. Ejemplo:
```
cd ..
```
Con este comando, se mueve al directorio Documents del usuario.
=====
```

Figura 92: Respuesta del modelo con la euclidea pero a la pregunta 'especializada'.

Esto ya es gusto del consumidor, ya que uno es directo (euclidea) y otro se explica algo más (coseno). Pero la coseno devuelve el contexto de donde saca la información.

12.1.7. Conclusiones de la pareja 'normal'

Vemos que, en general, rinde bastante bien, solo que con la coseno y pocos chunks sabe responder bien y dar el contexto de donde saca la información (tanto para preguntas normales como con código). Veremos si la pareja 'especial' es capaz de superarla notoriamente.

12.2. JINA y PHI: Los modelos 'especiales'

Veremos ahora si estos modelos, que han sido entrnados para entender código, pueden rendir mejor que la pareja normal (al menos, el la parte de código). Las preguntas serán las mismas que la pareja normal.



```
# Cargamos los modelos generativos a probar: phi
nombre_phi = "microsoft/Phi-3.5-mini-instruct"

# Cuantización en 4-bit para reducir VRAM (sino nos explota)
phi_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype="float16"
)

# Sacamos su tokenizador (lo que en verdad nos importa)
token_phi = AutoTokenizer.from_pretrained(nombre_phi)

# Y el modelo
model_phi = AutoModelForCausalLM.from_pretrained(
    nombre_phi,
    quantization_config=phi_config,
    device_map="auto" # reparte capas entre GPU y CPU
)

# Dejamos el pipeline preparado
phi = pipeline(
    "text-generation",
    model=model_phi,
    tokenizer=token_phi,
    device_map="auto"
)
```

Could not render content for 'application/vndupyter.widget-view+json'
`{ "model_id": "0f47aef2c2d452a56800fb27583859", "version_major": 2, "version_minor": 0 }`
Device set to use cuda: 0

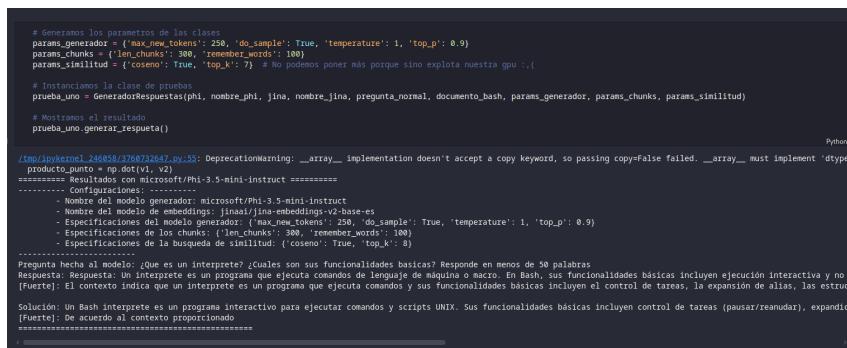
(a) Cargamos el modelo generador de embeddings: JINA.

(b) Cargamos el modelo generador de texto: PHI. Se carga así para cuantizarlo de tal manera que podamos usarlo.

Figura 93: Carga de los modelos 'normales'.

12.2.1. Probando con similitud coseno, con chunk de tamaño 300, remember de 100 y top_k=10

La prueba básica, igual que la primera pareja. Este es el resultado:



```
# Generamos los parametros de las clases
params_generador = {'max_new_tokens': 256, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
params_chunks = {'max_chunks': 300, 'remember_words': 100}
params_similitud = {'cosine': True, 'top_k': 7} # No podemos poner más porque sino explota nuestra gpu ;(
```

```
# Instanciamos la clase de pruebas
prueba_uno = GeneradorResuestas(phi, nombre_phi, jina, nombre_jina, pregunta_normal, documento_bash, params_generador, params_chunks, params_similitud)
```

```
# Mostramos el resultado
prueba_uno.generar_respueta()
```

----- Configuraciones -----
Nombre del modelo generador: microsoft/Phi-3.5-mini-instruct
Nombre del modelo de embeddings: jina/jina-embeddings-v2-base-4
Especificaciones de la generación: {'max_new_tokens': 256, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
Especificaciones de los chunks: {'max_chunks': 300, 'remember_words': 100}
Especificaciones de la búsqueda de similitud: {'cosine': True, 'top_k': 8}

Pregunta hecha al modelo: ¿Qué es un intérprete? ¿Cuáles son sus funcionalidades básicas? Responde en menos de 50 palabras
Respuesta: Respuesta: Un intérprete es un programa que ejecuta comandos de lenguaje de máquina o macro. En Bash, sus funcionalidades básicas incluyen ejecución interactiva y no [Fuerite]: El contexto indica que un intérprete es un programa que ejecuta comandos y sus funcionalidades básicas incluyen el control de tareas, la expansión de alias, las estructuras de datos y la ejecución de scripts.

Solución: Un Bash interprete es un programa interactivo para ejecutar comandos y scripts UNIX. Sus funcionalidades básicas incluyen control de tareas (pausar/reanudar), expandir variables y ejecutar scripts.

Figura 94: Respuesta del modelo especial con la similitud coseno.

Lo hace casi igual que la primera pareja (por no decir igual), solo que al menos trata de dar el chunk de donde lo sacó, pero no lo da bien (al parecer, solo da el chunk, pero no el número del chunk).

12.2.2. Probando con similitud euclidea, con chunk de tamaño 300, remember de 100 y top_k=10

Cambiamos la similitud de coseno a euclidea. Este es el resultado:

```
Vamos a ver como lo va con la distancia euclidea:

# Generamos los parametros de las clases
params_generador = {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
params_chunks = {'len_chunks': 300, 'remember_words': 100}
params_similitud = {'coseno': False, 'top_k': 7} # No podemos poner más porque sino explota nuestra gpu :(
# Instanciamos la clase de pruebas
prueba_dos = GeneradorResuestas(phi, nombre_phi, jina, nombre_jina, pregunta_normal, documento_bash, params_generador, params_chunks, params_similitud)
# Mostramos el resultado
prueba_dos.generar_respueta()

=====
Resultados con microsoft/Phi-3.5-mini-Instruct =====
----- Configuraciones: -----
Nombre del modelo generador: microsoft/Phi-3.5-mini-Instruct
Nombre del modelo de embeddings: Jina/linen-embedding-v2-base-es
- Especificaciones del modelo generador: {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
- Especificaciones de los chunks: {'len_chunks': 300, 'remember_words': 100}
- Especificaciones de la búsqueda de similitud: {'coseno': False, 'top_k': 7}
-----
Pregunta hecha al modelo: ¿Qué es un intérprete? ¿Cuáles son sus funcionalidades básicas? Responde en menos de 50 palabras
Respuesta: Respuesta: Un intérprete es un programa que ejecuta comandos y scripts letra por letra, conectando salida e entrada. En Bash, funcionalidades interactivas incluyen comandos de contexto:
- Capítulo 3: Funcionalidades Básicas del Intérprete (Sección 3.1)
- Capítulo 3: Sintaxis del Intérprete (Sección 3.2)
- Capítulo 3: Expansiones y Redirecciones (Sección 3.5)
- Capítulo 3: Control de Tareas y Bucles (Sección 3.6)
- Capítulo 3: Sintaxis y Expansiones (Sección 3.1)
-----
Explicación:
El contexto mencionado destaca las funcionalidades básicas del intérprete Bash, destacando su capacidad para ejecutar comandos linealmente, seguidos de operadores de control como
=====
```

Figura 95: Respuesta del modelo especial con la similitud euclidea.

Creemos que lo da algo incluso mejor que la pareja normal, pero no nos gusta que nos de contextos del texto, pero no de los chunks. A lo mejor, somos muy restrictivos, pero ese fallo no nos ha gustado.

12.2.3. Omitimos las pruebas con los chunks

No podemos jugar mucho con los chunks, ya que, los modelos, incluso cuantizados (y aplicando las optimizaciones de memoria que hemos podido), ocupan mucha VRAM, y nos es imposible jugar con los chunks.

12.2.4. Conclusiones con la pregunta normal (modelos especiales)

No se ven muchas diferencias, solo que intenta no fallar con dar el contexto (da los fragmentos, pero queremos el número). De momento, como mucho, la pareja 'normal' puede salir ganando al dar mejores explicaciones.

12.2.5. Probando ahora con la pregunta especializada (modelos especiales)

Ahora, usaremos los modelos para que nos respondan con la pregunta que habla de código:

1. Para la similitud coseno: Al no tener más opciones para esta similitud, este es el resultado:

```

# Generamos los parametros de las clases
params_generador = {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
params_chunks = {'len_chunks': 300, 'remember_words': 100}
params_similitud = {'coseno': True, 'top_k': 7} # No podemos poner más porque sino explota nuestra gpu :-(

# Instanciamos la clase de prueba
prueba_codigo_coseno = GeneradorRespuestas(phi, nombre_phi, jina, nombre_jina, pregunta_codigo, documento_bash, params_generador, params_chunks, params_similitud)

# Mostramos el resultado
prueba_codigo_coseno.generar_resputa()

[/tmp/tinykernel_246d98/37e0732a47.py:55]: DeprecationWarning: __array__ implementation doesn't accept a copy keyword, so passing copy=False failed. __array__ must implement __dtype__
produced by /opt/v1_v2
===== Resultados con microsoft/Phi-3.5-mini-instruct =====
----- Configuraciones: -----
- Nombre del modelo generador: microsoft/Phi-3.5-mini-instruct
- Nombre del modelo de embeddings: jina-embedding-densecosine-similarity
- Especificaciones del generador: {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
- Especificaciones de los chunks: {'len_chunks': 300, 'remember_words': 100}
- Especificaciones de la búsqueda de similitud: {'coseno': True, 'top_k': 7}
----- Pregunta hecha al modelo: ¿Para que sirve el comando "cd"? Puedes poner un ejemplo? Responde en menos de 50 palabras
Resputa: Output: El comando "cd" cambia el directorio de trabajo actual. Ejemplo: "cd /usr/bin" Cambia el directorio de trabajo actual al directorio /usr/bin.
Contexto: 208.- es llamado sin opciones), empleando por cero.-N Muestra el directorio número N (contando desde la derecha de la lista imprimida por dirs cuando es llamado sin opciones).
=====

```

Figura 96: Respuesta del modelo con la coseno pero a la pregunta 'especializada'.

2. Para la similitud euclidea: Pasa lo mismo aquí, así que, este es el resultado:

```

# Generamos los parametros de las clases
params_generador = {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
params_chunks = {'len_chunks': 300, 'remember_words': 100}
params_similitud = {'coseno': False, 'top_k': 7} # No podemos poner más porque sino explota nuestra gpu :-(

# Instanciamos la clase de prueba
prueba_codigo_euclidea = GeneradorRespuestas(phi, nombre_phi, jina, nombre_jina, pregunta_codigo, documento_bash, params_generador, params_chunks, params_similitud)

# Mostramos el resultado
prueba_codigo_euclidea.generar_resputa()

===== Resultados con microsoft/Phi-3.5-mini-instruct =====
----- Configuraciones: -----
- Nombre del modelo generador: microsoft/Phi-3.5-mini-instruct
- Nombre del modelo de embeddings: jina-embedding-densecosine-similarity
- Especificaciones del generador: {'max_new_tokens': 250, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
- Especificaciones de los chunks: {'len_chunks': 300, 'remember_words': 100}
- Especificaciones de la búsqueda de similitud: {'coseno': False, 'top_k': 7}
----- Pregunta hecha al modelo: ¿Para que sirve el comando "cd"? Puedes poner un ejemplo? Responde en menos de 50 palabras
Resputa: Output: El comando "cd" cambia el directorio actual. Ejemplo: Si quieres cambiar de directorio de usuario al directorio Documentos, ejecuta 'cd Documentos'. Output en formato de referencia: [208.-] Capítulo 6: Funcionalidades de Bash 103 dir
=====

```

Figura 97: Respuesta del modelo con la euclidea pero a la pregunta 'especializada'.

Vemos que, a lo mejor, lo hace mejor en general, ya que tanto con la similitud euclidea y coseno, es capaz de responder bien, y encima diciendo los fragmentos (chunks) de donde lo sacó.

12.2.6. Conclusiones de la pareja 'especial'

Vemos que rindió bastante bien, casi igual (e incluso algo mejor con la pregunta de código) que la pareja 'normal'. Como dijimos, era esperable que funcionara mejor con las preguntas de código.

12.3. Conclusiones generales

Las 2 parejas funcionan bastante bien si les ponemos la configuración adecuada. En general, creemos que la similitud coseno funciona un poquito mejor que la euclidea, aunque al final, si alguien prefiere esta última, es muy aceptable. Para nosotros, preferimos para cada pareja **la similitud coseno y sin cambiar los chunks**, ya que mostraron responder bien, y decir los chunks de donde sacaron la información (aunque es verdad que la última pareja, de momento, parece hacerlo algo mejor esto último).

13. Generando respuestas con ambas parejas

Para terminar (casi) esta memoria extensa, vamos a generar respuestas a unas cuantas preguntas, probando ambas parejas, y daremos una conclusión final.

```
# Definimos las preguntas
preguntas = [
    "¿Que hace el comando grep en Bash?",
    "¿Para que se utiliza awk en procesamiento de texto?",
    "¿Como funciona el comando sed para reemplazar cadenas?",
    "¿Cuales son las caracteristicas de BASH?",
    "¿Qual es la diferencia entre grep y awk?",
    "¿Para qué sirve el comando umask en Bash?"
]
```

Figura 98: Preguntas que tendrán que responder ambos modelos.

Nota: Dejaremos en un txt las respuestas de los modelos, ya que no queremos llenar la memoria solo de respuestas.

13.1. Respuestas de la pareja 'normal'

Estas fueron las respuestas de la pareja 'normal' (QWEN y MPNET):

```
# Generamos la respuesta con cada pregunta
for pregunta in preguntas:

    # Indicamos que es respuesta a una pregunta
    print(" " * 30, f'RESPUESTA A LA PREGUNTA "{pregunta}"', " " * 10)

    # Instanciamos la clase de pruebas
    generador_respuesta_especial = GeneradorRespuestas(phi, nombre_phi, jina, nombre_jina, pregunta, documentos_bash,
                                                       params_generador, parmas_chunks, params_similitud)

    # Mostramos el resultado
    generador_respuesta_especial.generar_respueta()

# PREGUNTA 1: RESPUESTA A LA PREGUNTA "¿Que hace el comando grep en Bash?"
# /home/jina/.local/lib/python3.5/site-packages/transformers/generation_utils.py:135: DeprecationWarning: ..._array implementation doesn't accept a copy keyword, so passing copy=False failed. ..._array must implement __dtypy_
producto_punto = np.dot(v1, v2)
=====
***** Resultados con microsoft/Phi-3.5-mini-Instruct *****
-----
Configuraciones:
- Nombre del modelo generador: microsoft/Phi-3.5-mini-instruct
- Nombre del modelo de embeddings: jina-jina-embeddings-v2-base-es
- Especificaciones del modelo generador: {'max_new_tokens': 256, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
- Especificaciones de los chunks: {'len_chunks': 300, 'remember_words': 100}
- Especificaciones de la búsqueda de similitud: {'cosine': True, 'top_k': 7}

Pregunta hecha al modelo: ¿Que hace el comando grep en Bash?
Respuesta: Respuesta:
El comando 'grep' en Bash se utiliza para buscar texto o patrones específicos dentro de archivos o entrada de flujo estándar. No daña ni modifica ningún archivo o entrada.

Con respecto a las respuestas y opciones detalladas en el contexto proporcionado:

- '-help' muestra un mensaje de uso y finaliza el comando, lo que indica que 'grep -h' o 'grep --help' proporciona ayuda interactiva.
- 'checkpoints' verifica si el programa o script es un archivo comprimido (.tar.gz) y no utiliza esta opción.
- 'checkpoints' es el equivalente de 'grep --checkpoints'. 'grep' no está relacionado con la gestión de tareas, por lo que esta opción no es relevante para 'grep'.
- 'checkwindow' ajusta el tamaño de la ventana de la consola para acordarse con el contenido de la tarea actual. 'grep' no ajusta vent

=====
***** PREGUNTA 2: RESPUESTA A LA PREGUNTA "¿Para que se utiliza awk en procesamiento de texto?"
***** Resultados con microsoft/Phi-3.5-mini-Instruct *****
-----
Configuraciones:
- Nombre del modelo generador: microsoft/Phi-3.5-mini-instruct
- Nombre del modelo de embeddings: jina-jina-embeddings-v2-base-es
- Especificaciones del modelo generador: {'max_new_tokens': 256, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
```

Figura 99: Parte de las respuestas de la pareja 'normal'.

Para ver las respuestas, pueden abrir el archivo 'respuestas_normal.txt', donde aparecen todas las respuestas. Diremos solo nuestras imprecisiones:

1. Responde a casi todas las preguntas, menos a la última (que es umask). Es verdad que no hay mucha información del comando, pero aún así la hay.
 2. Contesta muy bien, dando en casi todas el contexto de donde saca las respuestas.

3. No se hace problemas al responder preguntas sobre comandos (codigo), como para preguntas normales (que es algo).

En general, un rendimiento muy bueno, y se nota que aquí, RAG funcionó muy bien.

13.2. Respuestas de la pareja 'especial'

Estas fueron las respuestas de la pareja 'especial' (JINA y PHI):

```

D> # Generamos la respuesta con cada pregunta
for pregunta in preguntas:
    # Indicamos que es respuesta a una pregunta
    print(f"\n{10*'-' * 10} RESPUESTA A LA PREGUNTA '{pregunta}' {10*'-' * 10}\n")
    # Instanciamos la clase de pregunta
    generador_respuesta_especial = GeneradorRespuestas(phi, nombre_phi, jina, nombre_jina, pregunta, documento_bash,
                                                          params_generador, params_chunks, params_stabilitud)

    # Mostramos el resultado
    generador_respuesta_especial.generar_respueta()

[10]
... [10] RESPUESTA A LA PREGUNTA "¿Que hace el comando grep en Bash?" [10]
[10] [redacted] /usr/bin/grep: warning: /usr/bin/grep: DeprecationWarning: __array__ implementation doesn't accept a copy keyword, so passing copy=False failed. __array__ must implement
[10] producto_punto = np.dot(v1, v2)
=====
===== Resultados con microsoft/Phi:3.5-mini-instruct =====
===== Configuraciones: =====
- Nombre del modelo generador: microsoft/Phi:3.5-mini-instruct
- Nombre del modelo embeddings: jinaai/jina-embeddings-v2-base-es
- Especificaciones del modelo generador: {'max_new_tokens': 256, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}
- Especificaciones de los chunks: {'len_chunks': 300, 'remember_words': 100}
- Especificaciones de la búsqueda: {'cosine': True, 'top_k': 7}
=====
===== Pregunta hecha al modelo: ¿Que hace el comando grep en Bash?
Respueta: Respueta:
El comando 'grep' en Bash se utiliza para buscar texto o patrones específicos dentro de archivos o entrada de flujo estándar. No daña ni modifica ningún archivo o entrada.

Con respecto a las capacidades y opciones detalladas en el contexto proporcionado:
- '-help' muestra un mensaje de uso y finaliza el comando, lo que indica que 'grep -h' o 'grep --help' proporciona ayuda interactiva.
- 'checkhash' verifica si un programa o script es un archivo comprimido. 'grep' no utiliza esta opción.
- 'checkjob' registra el estado de las tareas del computo. 'grep' no está relacionado con la gestión de tareas, por lo que esta opción no es relevante para 'grep'.
- 'checksize' ajusta el tamaño de los resultados de la consola para acordar con el contenido de la tarea actual. 'grep' no ajusta venta
=====
===== RESPUESTA A LA PREGUNTA "¿Para qué se utiliza awk en procesamiento de texto?" =====
===== Resultado con microsoft/Phi:3.5-mini-instruct =====
- Nombre del modelo generador: microsoft/Phi:3.5-mini-instruct
- Nombre del modelo de embeddings: jinaai/jina-embeddings-v2-base-es
- Especificaciones del modelo generador: {'max_new_tokens': 256, 'do_sample': True, 'temperature': 1, 'top_p': 0.9}

```

Figura 100: Parte de las respuestas de la pareja 'especial'.

Para ver las respuestas, pueden abrir el archivo 'respuestas_especial.txt', donde aparecen todas las respuestas. Diremos solo nuestras imprecisiones:

1. No responde a todas las preguntas. Solo responde a la de 'grep', 'sed' y 'diferencia entre grep y awk', llegando incluso a responder sobre 'sed' con su propio conocimiento (cuando en verdad, el manual tiene las respuestas a las preguntas, solo que a veces están 'rebuscadas'). Esto seguramente se debe a que no tiene los mismos chunks que la pareja 'normal', al limitarlo por su gran consumo de VRAM.
2. Quitando esto, contesta bien las preguntas que respondió (y contesta la que no pudo responder la pareja 'normal': La de umask).
3. No se ven muchas diferencias entre las respuestas a código (comandos) con respecto a la pareja normal (nos referimos a diferencias significativas, rollo de mejor explicado y tal).

En general, se entiende que se limita por no tener la misma información (shunks) que la pareja 'normal'. Quitando eso, se puede decir que lo hace bien, casi igual que la pareja 'normal'.

13.3. Conclusiones finales de las respuestas

Al final, para nosotros, la pareja normal respondió mucho mejor que la pareja especial, ya que, siendo sinceros, el que la pareja 'especial' no tenga la misma información (chunks), es una gran desventaja l no contar con información suficiente para responder. Aún así, recordemos que, con la pareja 'normal', al aumentar los chunks, tuvo un colapso, por lo cual, a lo mejor esto influye hasta un punto. Luego, la pareja 'especial', no es capaz de buscar bien en el documento la información para responder las preguntas, cosa que si hizo la pareja 'normal', fallando en una sola pregunta. Como observación final, recomendamos usar a la pareja 'normal', ya que, primero que todo, no consume muchos recursos, son bastante livianos. En segundo lugar, respondieron mejor a las preguntas finales. Por último, sin necesidad de pasar mucha información (20 chunks de gran tamaño: 500 palabras con 200 de memoria), es capaz de responder bien. Para terminar, usad esta pareja, ya que funciona muy bien (y consume no mucho comparado con otros modelos). Solo nos queda la espina de no arreglar como saca las respuestas, ya que a veces las saca algo cortas o largas. No probamos en aumentar los tokens por el miedo del OutOfMemory. Si alguno cuenta con una mejor gráfica que la RTX 4060 de portátil, puede probar esto, que a lo mejor rinde mucho mejor.

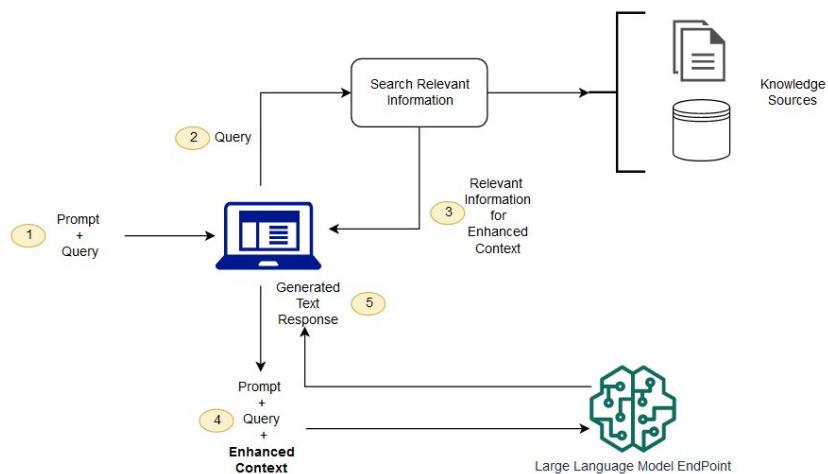


Figura 101: Imagen de como funciona RAG.

14. Conclusiones finales

Tanot RAG como LoRA, son técnicas potentes para coger modelos pre-entrenados y especializarlos para cualquier tarea que queramos. A lo mejor, LoRA dió la impresión de funcionar mejor, pero creemos que ambas son muy útiles en general (es más, vimos que en la vida real, los chatbots usan el principio de RAG para ello). Solo es contar con algo de hardware, ponerse a probar y saber elegir bien los modelos, ya que si se elige mal, por mucho que se entrene bien, fallará en dar las respuestas al no ser los indicados.



Figura 102: Imagen de Gemini sobre LoRA y RAG.

15. Referencias

1. **Paper de Deberta:** <https://arxiv.org/abs/2111.09543>
2. **Paper de Roberta:** <https://arxiv.org/abs/1907.11692>
3. **Ayuda de ChatGPT (para algunas dudas):** <https://chatgpt.com/>
4. **Ayuda de Copilot, para algunas cosas de programar:** <https://copilot.microsoft.com/>
5. **Ayuda de Gemini, para hacer el Latex:** <https://gemini.google.com/app>