Ajuste fino de DistilBERT para Tareas de Elección Múltiple

Autor: Jorge Elías García
Correo: jorge.elias@alumnos.upm.es

Introducción

En este cuaderno se detallará el proceso de seleccionar, configurar y ejecutar el ajuste fino (fine-tuning) de un Modelo de Lenguaje (LM) preentrenado para una tarea específica de Procesamiento del Lenguaje Natural (NLP). Concretamente, se realizará un **fine-tunning para *Multiple Choice* empleando DistilBERT como LLM preentrenado**.

En primer lugar, abordamos una tarea de **NLP de elección múltiple**, concretamente un problema en el que el modelo debe seleccionar la continuación más coherente de un contexto entre varias opciones posibles. Este tipo de tareas exige que el modelo no solo comprenda el contenido explícito del texto, sino que también sea capaz de razonar sobre relaciones causales, semánticas y de sentido común. Para ello utilizamos el dataset **SWAG**, un conjunto de ejemplos diseñado específicamente para evaluar la capacidad de los modelos de lenguaje en tareas de razonamiento situacional (*commonsense reasoning*).

En segundo lugar, para esta tarea se ha elegido `distilbert-base-uncased` como modelo de lenguaje base preentrenado, ya que se trata de una versión eficiente y eficaz de la familia BERT, manteniendo un **rendimiento similar al de BERT**, pero con **menos parámetros**, lo que conlleva tiempos de entrenamiento más reducidos. Asimismo, al ser una arquitectura *encoder-only*, está específicamente diseñada para tareas de **comprensión** y **razonamiento** sobre el texto, lo que resulta especialmente adecuado para el conjunto **SWAG**, donde el modelo debe analizar un contexto y determinar cuál de las opciones es la continuación más plausible. Por estos motivos, se ha considerado **DistilBERT** como la arquitectura *encoder-only* más práctica y equilibrada para llevar a cabo el *fine-tuning* en una tarea de *multiple choice* sin sacrificar calidad en la predicción.

En último lugar, dado que `distilbert-base-uncased` no está entrenado para tareas de *multiple choice*, es necesario emplear la técnica de **ajuste fino (fine-tuning)** sobre SWAG. El proceso de ajuste fino consiste en adaptar DistilBERT a esta tarea mediante varios pasos encadenados: la carga y tokenización del dataset SWAG, la preparación del modelo con una cabeza específica de *multiple choice*, la definición de los hiperparámetros de entrenamiento, la implementación de una función para la evaluación del modelo y, finalmente, el entrenamiento supervisado junto con la evaluación periódica del rendimiento. Este flujo permite especializar el modelo en seleccionar la continuación correcta entre varias opciones a partir del contexto proporcionado.

Carga de las librerías necesarias y comprobación de GPU

En primer lugar, cargamos las librerías necesarias. Incorporaremos:

- `transformers`: Colección de modelos de lenguaje preentrenados (e.g., BERT) para tareas de NLP. Además de herramientas necesarias, como el tokenizador.
- `datasets`: Acceso simple a grandes datasets.
- `evaluate`: Herramienta que ofrece las métricas esenciales (como accuracy) para evaluar el rendimiento del modelo.
- `accelerate`: Permite un entrenamiento rápido y optimizado en diferentes configuraciones de hardware (CPU/GPU).

```
!pip install -q transformers datasets accelerate evaluate
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84.1/84.1 kB 3.9 MB/s eta 0:00:00
```

Comprobamos también la disponibilidad de GPU.

```
import torch

if torch.cuda.is_available():
    print(f"GPU detected: {torch.cuda.get_device_name(0)}")
else:
    print("No GPU detected.")
    print("If you are using Google Colab, please go to 'Runtime' > 'Change runtime type' and select 'GPU' as the hardware a
```

```
GPU detected: Tesla T4
```

Dataset

El dataset `swag` (*Situations With Adversarial Generations*) es un corpus diseñado para evaluar **razonamiento de sentido común** mediante tareas de **selección múltiple**. Cada ejemplo contiene un breve contexto dividido en dos partes (`sent1` y `sent2`) y **cuatro posibles continuaciones** (`ending0`, `ending1`, `ending2`, `ending3`). El objetivo del modelo consiste en predecir cuál de estas opciones constituye la **continuación más plausible** según el sentido común.

Usamos la librería `datasets` para descargar el conjunto de datos.

```
from datasets import load_dataset

# Cargamos la versión estándar ("regular") del dataset SWAG.
dataset = load_dataset("swag", "regular")

# Comprobamos el tamaño de los conjuntos de entrenamiento y validación.
print("Train:", len(dataset["train"]))
print("Validation:", len(dataset["validation"]))
print("Test:", len(dataset["test"]))

# Mostramos un ejemplo.
ej = dataset["train"][0]
print("sent1:", ej["sent1"])
print("sent2:", ej["sent2"])
print("\nOpciones:")
for i in range(4):
    print(f"  ending{i}:", ej[f"ending{i}"])
print("\nLabel correcto:", ej["label"])
```

```
Train: 73546
Validation: 20006
Test: 20005
sent1: Members of the procession walk down the street holding small horn brass instruments.
sent2: A drum line

Opciones:
  ending0: passes by walking down the street playing their instruments.
  ending1: has heard approaching them.
  ending2: arrives and they're outside dancing and asleep.
  ending3: turns the lead singer watches the performance.

Label correcto: 0
```

Definimos desde ya el nombre del modelo, pues será necesario para el tokenizador.

```
model_name = "distilbert-base-uncased"
```

Preparación del dataset

Para preparar los datos del conjunto **SWAG** se utiliza la clase `AutoTokenizer`, que permite cargar automáticamente el tokenizador específico del modelo preentrenado. Esto asegura que el texto se procese con el **mismo vocabulario y las mismas reglas de segmentación** que el modelo escogido, manteniendo la coherencia entre los tokens generados y los embeddings que el modelo espera recibir.

La tokenización implica **dividir el texto en subpalabras o tokens**, convertir esos tokens en **índices numéricos**, y unificar la longitud de todas las secuencias mediante **padding** o **truncación** cuando es necesario. Además, se añaden los **tokens especiales** propios del modelo y se construyen **máscaras de atención** que distinguen entre contenido real y posiciones de relleno. Con esta representación numérica uniforme, el modelo puede interpretar correctamente cada par *(contexto, opción)*.

```
from transformers import AutoTokenizer

# Descargamos el tokenizador.
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Función de preprocesado / tokenización para Multiple Choice.
def preprocess_function(examples):
    # Construimos el contexto concatenando sent1 + sent2.
    contexts = [s1 + " " + s2 for s1, s2 in zip(examples["sent1"], examples["sent2"])]

    # SWAG tiene 4 opciones por ejemplo.
    choices = list(zip(
        examples["ending0"],
        examples["ending1"],
        examples["ending2"],
        examples["ending3"]
    ))
```

```
    # Aplanamos listas para tokenizar (contexto repetido 4 veces por ejemplo).
    contexts_flat = []
    choices_flat = []

    for ctx, four_endings in zip(contexts, choices):
        for ending in four_endings:
            contexts_flat.append(ctx)
            choices_flat.append(ending)

    # Tokenización batch con el tokenizador de DistilBERT.
    tokenized = tokenizer(
        contexts_flat,
        choices_flat,
        truncation=True,
        padding="max_length",
        max_length=128,
    )

    # Volver a agrupar a forma (batch_size, num_choices=4, seq_len).
    num_choices = 4
    result = {}
    for key, val in tokenized.items():
        # val es una lista plana; la rehacemos en bloques de 4.
        result[key] = [val[i:i+num_choices] for i in range(0, len(val), num_choices)]

    # Añadimos las etiquetas
    result["labels"] = examples["label"]
    return result


# Creamos subconjuntos pequeños en texto y posteriormente tokenizamos solo esos subcojuntos, para mayor eficiencia.
small_train_raw = dataset["train"].shuffle(seed=42).select(range(1000))
small_eval_raw  = dataset["validation"].shuffle(seed=42).select(range(300))
small_train_dataset = small_train_raw.map(
    preprocess_function,
    batched=True,
    remove_columns=small_train_raw.column_names  # limpia columnas originales.
)

small_eval_dataset = small_eval_raw.map(
    preprocess_function,
    batched=True,
    remove_columns=small_eval_raw.column_names
)

# Imprimimos un ejemplo ya tokenizado.
print(small_train_dataset[0])
```

```
tokenizer_config.json: 100%                                  48.0/48.0 [00:00<00:00, 1.74kB/s]
config.json: 100%                                            483/483 [00:00<00:00, 11.1kB/s]
vocab.txt: 100%                                              232k/232k [00:00<00:00, 1.88MB/s]
tokenizer.json: 100%                                         466k/466k [00:00<00:00, 3.57MB/s]
Map: 100%                                                    1000/1000 [00:01<00:00, 736.71 examples/s]
Map: 100%                                                    300/300 [00:00<00:00, 720.22 examples/s]
{'input_ids': [[101, 2002, 4084, 2041, 1997, 1996, 6080, 4576, 1998, 14020, 4487, 19022, 20806, 23559, 2135, 1012, 1999, 199
```

Implementación

A continuación, se detalla el procedimiento de carga del LM, detalles de los hiperparámetros de entrenamiento y ejecución del ajuste fino.

Reemplazamos la cabeza del modelo

Para adaptar *DistilBERT* a la tarea de *multiple choice* del dataset **SWAG**, sustituimos su cabeza original por una cabeza especializada para la selección múltiple mediante la clase `AutoModelForMultipleChoice` de la librería **Transformers** (HuggingFace). Esta variante del modelo añade una capa final diseñada específicamente para puntuar varias opciones simultáneamente. En nuestro caso, indicamos `num_labels=4` porque cada ejemplo de SWAG contiene exactamente **cuatro posibles continuaciones del contexto**.

Esta cabeza toma las representaciones generadas por DistilBERT para cada par *(contexto, opción)*, calcula un **logit** independiente para cada una de ellas y selecciona como predicción la opción con puntuación más alta. De este modo, en lugar de asignar una etiqueta fija a un texto, el modelo compara opciones entre sí bajo un mismo contexto.

```
from transformers import AutoModelForMultipleChoice

# Cargamos el modelo con cabeza de multiple choice (4 opciones en SWAG).
model = AutoModelForMultipleChoice.from_pretrained(model_name, num_labels=4)
```

```
model.safetensors:  100%                                                268M/268M  [00:03<00:00,  128MB/s]
Some weights of DistilBertForMultipleChoice were not initialized from the model checkpoint at distilbert-base-uncased and ar
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

Seleccionamos los hiperparámetros para el entrenamiento

Para controlar el proceso de *fine-tuning*, empleamos la clase `TrainingArguments` de la librería **Transformers** (HuggingFace). Esta interfaz nos permite definir los principales hiperparámetros del entrenamiento, que luego serán utilizados por el `Trainer`.

Entre los parámetros que se pueden configurar se encuentran:

- `eval_strategy`
  Determina cuándo se realiza la evaluación del modelo (por ejemplo, por épocas o por pasos).
- `learning_rate`
  Controla la magnitud de las actualizaciones de los pesos durante el entrenamiento.
- `num_train_epochs`
  Indica cuántas veces el modelo recorre completamente el conjunto de entrenamiento.
- `weight_decay`
  Forma de regularización que penaliza pesos demasiado grandes para evitar sobreajuste.
- `report_to`
  Especifica a qué herramientas externas se envían los registros de entrenamiento. En nuestro caso lo desactivamos (`"none"`) para evitar cualquier integración automática con plataformas como Weights & Biases, lo cual impide que durante el entrenamiento aparezcan avisos pidiendo claves o configuraciones externas que no necesitamos.

Estos han sido los hiperparámetros seleccionados. No obstante, existen otras muchas variables con posibilidad de configuración, para de esta manera adaptar el entrenamiento a los requerimientos específicos del usuario.

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    eval_strategy="epoch", # La evaluación del modelo se hará al acabar cada época.
    learning_rate=2e-5, # Damos un valor bajo al learning rate, para evitar cambios muy bruscos u olvido catastrófico.
    num_train_epochs=3, # Entrenamos durante 3 épocas, para prevenir el olvido catastrófico.
    weight_decay=0.01, # Penalizamos los pesos muy grandes para evitar sobreajuste.
    report_to="none" # Descativamos los loggings externos.
)
```

Creamos una función para la evaluación del modelo

El `Trainer` no calcula métricas por sí mismo, por lo que necesitamos definir una función externa que indique cómo evaluar el rendimiento del modelo. Para ello utilizamos la librería **Evaluate** de HuggingFace, que ofrece implementaciones listas para usar de métricas habituales, como la *accuracy*.

Primero cargamos la métrica con `evaluate.load`, y luego definimos una función `compute_metrics` que el `Trainer` llamará automáticamente cada vez que corresponda realizar una evaluación. Esta función recibe las predicciones del modelo y las etiquetas reales, extrae la clase más probable en cada ejemplo y devuelve el valor de la métrica seleccionada.

La frecuencia con la que esta evaluación ocurre depende del parámetro `evaluation_strategy` definido en `TrainingArguments`, que puede indicar, por ejemplo, que la evaluación se realice al finalizar cada época (como en nuestro caso).

```
import evaluate

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

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = logits.argmax(axis=-1)
    return accuracy_metric.compute(predictions=predictions, references=labels)
```

```
Downloading builder script:       4.20k/? [00:00<00:00, 379kB/s]
```

Entrenamos y evaluamos el modelo

Finalmente, podemos hacer uso de la clase `Trainer` para llevar a cabo el entrenamiento del modelo. Este componente unifica todo el proceso de entrenamiento y evaluación, permitiendo ejecutar el pipeline completo de forma automática.

El `Trainer` recibirá:

- `model`: el modelo ya configurado con la cabeza de *multiple choice*.
- `args`: los hiperparámetros definidos previamente mediante `TrainingArguments`.
- `train_dataset` y `eval_dataset`: los subconjuntos de entrenamiento y validación ya preprocesados.
- `compute_metrics`: la función encargada de calcular las métricas durante la evaluación.

Una vez inicializado, simplemente llamaremos a `trainer.train()` para iniciar el proceso de *fine-tuning* de principio a fin.

```
from transformers import Trainer

# Empleamos todos los parámetros creados previamente.
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=small_train_dataset,
    eval_dataset=small_eval_dataset,
    compute_metrics=compute_metrics,
)


trainer.train()
```

[375/375 02:13, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy
1	No log	1.287296	0.430000
2	No log	1.162687	0.473333
3	No log	1.169987	0.503333

```
TrainOutput(global_step=375, training_loss=1.091301513671875, metrics={'train_runtime': 134.4219,
'train_samples_per_second': 22.318, 'train_steps_per_second': 2.79, 'total_flos': 397395108864000.0, 'train_loss':
1.091301513671875, 'epoch': 3.0})
```

Resultados y Discusión

A lo largo del proceso de *fine-tuning*, el modelo muestra una evolución progresiva tanto en la pérdida de validación como en la métrica de *accuracy*. Aunque el conjunto de datos empleado es relativamente reducido, se aprecia una mejora estable entre épocas. En la tercera y última época, el modelo alcanza aproximadamente un **50% de acierto**, lo que refleja una capacidad razonable para seleccionar continuaciones coherentes en la tarea de *multiple choice*, especialmente teniendo en cuenta la complejidad semántica del dataset SWAG y el tamaño limitado del subconjunto utilizado.

Es importante considerar que modelos como DistilBERT pueden **sobreajustarse rápidamente** cuando trabajan con cantidades pequeñas de datos. Incrementar el número de épocas podría mejorar temporalmente el rendimiento, pero también aumenta el riesgo de *overfitting* o incluso de **olvido catastrófico**, donde el modelo pierde parte del conocimiento adquirido durante su preentrenamiento al especializarse en exceso en la tarea actual. Por este motivo no ampliamos el número de épocas: mantener el entrenamiento acotado ayuda a evitar que el modelo se sobreajuste o pierda parte del conocimiento adquirido durante el preentrenamiento, preservando así un equilibrio adecuado entre aprendizaje y generalización.

Predicción con el modelo ajustado

Para evaluar el modelo sobre ejemplos personalizados de *multiple choice* seguimos la misma lógica que durante el entrenamiento, pero aplicada a nuevos datos. Es fundamental reutilizar **el mismo tokenizador** y **el mismo modelo entrenado**, de modo que el texto se procese con el mismo vocabulario, los mismos tokens especiales y el mismo formato de entrada que el modelo ha visto durante el *fine-tuning*.

Cada ejemplo se construye combinando un **mismo contexto** con sus **cuatro opciones de respuesta**, que el tokenizador transforma en pares `(contexto, opción)` y convierte en tensores listos para el modelo. Para cada una de estas secuencias, DistilBERT genera una representación contextualizada (*embedding*) que resume la información del par completo. Estas cuatro representaciones llegan a la cabeza de *multiple choice*, que calcula un **logit** para cada opción. Finalmente, el modelo devuelve los loggits y seleccionamos la respuesta con la puntuación más alta, equivalente a la opción que el modelo considera más probable.

Ejecutamos `model.eval()` al comienzo de la celda, para cerciorarnos de que el modelo se encuentra en modo inferencia.

```
model.eval() # Aseguramos de que el modelo esté en modo de evaluación.

def predict_multiple_choice(custom_sent1, custom_sent2, custom_options, tokenizer, model, device, max_length=64):
    """
    Realizamos predicciones sobre ejemplos personalizados de multiple choice.

    custom_sent1: primera oración del contexto.
    custom_sent2: segunda oración del contexto.
    custom_options: lista de listas, cada sublista contiene las 4 opciones.
    tokenizer: tokenizador del modelo.
    model: modelo entrenado.
    device: "cuda" o "cpu".
    max_length: longitud máxima de tokenización.
    """

    # Preparamos listas planas para tokenización (igual que en preprocess_function).
    contexts_flat = []
    choices_flat = []

    for s1, s2, opts in zip(custom_sent1, custom_sent2, custom_options):
        ctx = s1 + " " + s2
        for opt in opts:
            contexts_flat.append(ctx)
            choices_flat.append(opt)

    # Tokenizamos.
    tokenized = tokenizer(
        contexts_flat,
        choices_flat,
        truncation=True,
        padding="max_length",
        max_length=max_length,
        return_tensors="pt"
    )

    # Reagrupamos a forma (batch_size, num_choices, seq_len).
    num_choices = 4
    input_ids = tokenized["input_ids"].view(-1, num_choices, tokenized["input_ids"].size(-1))
    attention_mask = tokenized["attention_mask"].view(-1, num_choices, tokenized["attention_mask"].size(-1))

    inputs = {
        "input_ids": input_ids.to(device),
        "attention_mask": attention_mask.to(device)
    }

    # Realizamos la inferencia.
    with torch.no_grad():
        outputs = model(**inputs)
        logits = outputs.logits
        preds = torch.argmax(logits, dim=-1).tolist()

    # Mostramos los resultados.
    for i, pred_idx in enumerate(preds):
        print(f"\n===== EJEMPLO {i} =====")
        print("Contexto:", custom_sent1[i], custom_sent2[i])
        print("\nOpciones:")
        for j, opt in enumerate(custom_options[i]):
            marcador = "  <-- PREDICHA" if j == pred_idx else ""
            print(f"  [{j}] {opt}{marcador}")
```

1. Predicción de ejemplos sencillos

En primer lugar, estudiamos el rendimiento del modelo con un conjunto de **ejemplos más sencillos.** En teoría, los ejemplos deberían ser suficientemente fáciles para que el modelo identifique claramente la opción que completa el contexto de manera más coherente. Consecuentemente, comprobamos si el modelo ha aprendido el patrón general de la tarea y es capaz de escoger la continuación más lógica antes de pasar a casos más complejos o ambiguos.

```
# Ejemplos personalizados.

device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # Si es posible empleamos la GPU.

custom_sent1 = [
    "He dropped the glass on the floor",
    "During her trip to Tenerife"
]
```

```
custom_sent2 = [
    "and then he",
    "she decided to"
]

custom_options = [
    [  # opciones para el ejemplo 0
        "picked it up carefully.",
        "watched it fly away into the sky.",
        "ignored the loud music.",
        "answered the phone."
    ],
    [  # opciones para el ejemplo 1
        "prepare a snowman in the icy mountains.",
        "ski across the frozen valley.",
        "visit the beach on the sunny coast.",
        "explore the deep Arctic glacier."
    ]
]

predict_multiple_choice(
    custom_sent1,
    custom_sent2,
    custom_options,
    tokenizer,
    model,
    device
)
```

```
===== EJEMPLO 0 =====
Contexto: He dropped the glass on the floor and then he

Opciones:
  [0] picked it up carefully.  <-- PREDICHA
  [1] watched it fly away into the sky.
  [2] ignored the loud music.
  [3] answered the phone.

===== EJEMPLO 1 =====
Contexto: During her trip to Tenerife she decided to

Opciones:
  [0] prepare a snowman in the icy mountains.
  [1] ski across the frozen valley.
  [2] visit the beach on the sunny coast.  <-- PREDICHA
  [3] explore the deep Arctic glacier.
```

Apreciamos que el modelo ha **razonado correctamente**, prediciendo la continuación más lógica para ambos ejemplos.

2. Predicción de ejemplos más complejos

Tras verificar el comportamiento del modelo en situaciones sencillas, analizamos cómo se comporta ante ejemplos menos directos o con opciones más similares entre sí. Este tipo de casos pone a prueba la capacidad del modelo para captar matices semánticos, resolver ambigüedades y seleccionar la opción más coherente cuando el contexto no ofrece una pista evidente. Evaluar ejemplos más complejos nos permite comprobar si el *fine-tuning* ha dotado realmente al modelo de una comprensión más profunda de la tarea, más allá de los patrones triviales.

```
# Ejemplos personalizados más complejos.
custom_sent1 = [
    "After running for two hours in the rain",
    "The child looked at the strange device on the table",
    "When the rocket finally launched into the sky",
    "After landing in Tenerife for the first time"
]

custom_sent2 = [
    "he finally",
    "and slowly",
    "the scientists in the control room",
    "she looked around and"
]

custom_options = [
    [  # Ejemplo 0
        "felt a wave of exhaustion hit him.",
        "started preparing a large meal.",
        "painted the walls of his house blue.",
```

```
        "won the lottery unexpectedly."
    ],
    [  # Ejemplo 1
        "reached for his backpack to go home.",
        "took a step back, unsure of what it did.",
        "flew out the window like a bird.",
        "began singing loudly in the street."
    ],

    [  # Ejemplo 2
        "watched with excitement as data streamed in.",
        "baked a cake to celebrate the event.",
        "took a nap on the floor.",
        "cleaned the windows in the building."

    ],
    [  # Ejemplo 3
        "admired the volcanic landscape and warm breeze.",
        "joined a high-speed car race across the mountains.",
        "painted a huge mural inside the airplane cabin.",
        "went directly to the Eiffel Tour."
    ]
]

predict_multiple_choice(
    custom_sent1,
    custom_sent2,
    custom_options,
    tokenizer,
    model,
    device
)
```

```
===== EJEMPLO 0 =====
Contexto: After running for two hours in the rain he finally

Opciones:
  [0] felt a wave of exhaustion hit him.
  [1] started preparing a large meal.  <-- PREDICHA
  [2] painted the walls of his house blue.
  [3] won the lottery unexpectedly.

===== EJEMPLO 1 =====
Contexto: The child looked at the strange device on the table and slowly

Opciones:
  [0] reached for his backpack to go home.
  [1] took a step back, unsure of what it did.  <-- PREDICHA
  [2] flew out the window like a bird.
  [3] began singing loudly in the street.

===== EJEMPLO 2 =====
Contexto: When the rocket finally launched into the sky the scientists in the control room

Opciones:
  [0] watched with excitement as data streamed in.  <-- PREDICHA
  [1] baked a cake to celebrate the event.
  [2] took a nap on the floor.
  [3] cleaned the windows in the building.

===== EJEMPLO 3 =====
Contexto: After landing in Tenerife for the first time she looked around and

Opciones:
  [0] admired the volcanic landscape and warm breeze.  <-- PREDICHA
  [1] joined a high-speed car race across the frozen mountains.
  [2] painted a huge mural inside the airplane cabin.
  [3] went directly to the Eiffel Tour
```

A partir de estos ejemplos más complejos, observamos que el modelo tiende a seleccionar opciones **coherentes con el contexto**, incluso cuando la respuesta correcta no es trivial. En el primer caso, aunque **no elige la opción que refleja de forma más directa el cansancio físico tras correr bajo la lluvia, sí selecciona una continuación razonable dentro de la situación**, lo que indica que mantiene cierta consistencia semántica. En los demás ejemplos, el modelo identifica correctamente la opción que mejor encaja: se aleja de alternativas absurdas o incompatibles y prefiere aquellas que guardan una relación lógica con la escena descrita, como la reacción cautelosa ante un dispositivo desconocido, el seguimiento de datos en un lanzamiento de cohete o la referencia al paisaje volcánico y clima cálido en Tenerife. En conjunto, estos resultados sugieren que **el modelo ha aprendido a utilizar el contexto para filtrar opciones y priorizar continuaciones plausibles.**

Evaluación del rendimiento y limitaciones

El modelo muestra un comportamiento razonablemente sólido tras el ajuste fino. En los ejemplos sencillos, donde la relación entre el contexto y la opción correcta es directa, el modelo no comete errores y selecciona siempre la continuación esperada. En los ejemplos más complejos —aquellos con alternativas plausibles o ambiguas— el modelo mantiene una elección "coherente", evitando opciones incompatibles o sin sentido. Aunque no siempre identifica la mejor respuesta, sí tiende a elegir opciones que encajan semánticamente con la situación descrita.

No obstante, las limitaciones también se hacen evidentes. Al evaluar el modelo sobre el conjunto completo de validación de SWAG, su precisión se sitúa en torno al **50%**, lo que indica a pensar que, con frecuencia, selecciona respuestas razonables pero no necesariamente la más adecuada. Esto refleja la dificultad inherente de la tarea, el tamaño reducido del conjunto utilizado para el entrenamiento y la propia capacidad de DistilBERT, que puede quedarse corto para capturar matices más sutiles de razonamiento de sentido común. En conjunto, aunque el modelo muestra un desempeño aceptable, aún queda margen de mejora para alcanzar una comprensión más fina del contexto.

Conclusión

En este trabajo hemos llevado a cabo el *fine-tuning* de **DistilBERT** para una tarea de *multiple choice* utilizando el dataset **SWAG**, siguiendo todas las etapas necesarias para este proceso: preprocesamiento del conjunto de datos, adaptación del modelo mediante la cabeza `AutoModelForMultipleChoice`, configuración estructurada de los hiperparámetros con `TrainingArguments`, integración de una función de evaluación personalizada y entrenamiento supervisado con el `Trainer`. Posteriormente, evaluamos el comportamiento del modelo tanto en ejemplos sencillos como en casos más complejos y ambiguos para valorar su capacidad de razonamiento contextual.

Los resultados muestran que, tras el *fine-tuning*, el modelo es capaz de **seleccionar continuaciones razonables en muchos casos, evitando opciones claramente incompatibles y manteniendo cierta coherencia contextual en ejemplos aislados**. Sin embargo, al evaluar su rendimiento en el propio conjunto SWAG, el modelo no supera el **50% de acierto**, lo que indica que tiene dificultades para identificar sistemáticamente la opción más adecuada cuando varias alternativas son plausibles. Esta limitación sugiere que la tarea plantea un nivel de complejidad que DistilBERT, incluso ajustado, no siempre logra resolver, especialmente en situaciones donde se requiere un razonamiento más fino o conocimiento implícito de sentido común.

Finalmente, este proceso pone de manifiesto la flexibilidad que ofrece la librería **Transformers** y el uso del **Trainer** basado en PyTorch. El *fine-tuning* completo puede implementarse con muy pocas líneas de código, pero al mismo tiempo permite un ajuste totalmente personalizado del modelo sobre **nuestros propios datos**, a diferencia del uso directo de modelos ya entrenados mediante la función `pipeline`. Asimismo, probar diferentes arquitecturas es muy ágil: cambiar de DistilBERT a RoBERTa o a cualquier otro modelo solo requiere ajustar muy pocas líneas de código, lo que facilita evaluar de forma inmediata cómo varía el rendimiento entre modelos.