Phase 1: Data Preprocessing

```
pip install tensorflow nltk scikit-learn
```

```
Requirement already satisfied: tensorflow in /usr/local/lib/python3.11/dist-packages (2.18.0)
Requirement already satisfied: nltk in /usr/local/lib/python3.11/dist-packages (3.9.1)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (1.6.1)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (25.2.10)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (0.6.0)
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (0.2.0)
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (18.1.1)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.4.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from tensorflow) (24.2)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<6.0.0dev,>=3.20.3 in /usr/local/lib/python
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-packages (from tensorflow) (75.2.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.17.0)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.0.1)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (4.13.1)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.17.2)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (1.71.0)
Requirement already satisfied: tensorboard<2.19,>=2.18 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (2.18.0)
Requirement already satisfied: keras>=3.5.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.8.0)
Requirement already satisfied: numpy<2.1.0,>=1.26.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (2.0.2)
Requirement already satisfied: h5py>=3.11.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (3.13.0)
Requirement already satisfied: ml-dtypes<0.5.0,>=0.4.0 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (0.4.1)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/lib/python3.11/dist-packages (from tensorflow) (0.37.1
Requirement already satisfied: click in /usr/local/lib/python3.11/dist-packages (from nltk) (8.1.8)
Requirement already satisfied: joblib in /usr/local/lib/python3.11/dist-packages (from nltk) (1.4.2)
Requirement already satisfied: regex>=2021.8.3 in /usr/local/lib/python3.11/dist-packages (from nltk) (2024.11.6)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from nltk) (4.67.1)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.14.1)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (3.6.0)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.11/dist-packages (from astunparse>=1.6.0->tensorflow) (0.45.
Requirement already satisfied: rich in /usr/local/lib/python3.11/dist-packages (from keras>=3.5.0->tensorflow) (13.9.4)
Requirement already satisfied: namex in /usr/local/lib/python3.11/dist-packages (from keras>=3.5.0->tensorflow) (0.0.8)
Requirement already satisfied: optree in /usr/local/lib/python3.11/dist-packages (from keras>=3.5.0->tensorflow) (0.14.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensorflow
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensorflow) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensorflow) (2.3
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests<3,>=2.21.0->tensorflow) (202
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.11/dist-packages (from tensorboard<2.19,>=2.18->tensorflow) (3.
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local/lib/python3.11/dist-packages (from tensorboard<2.19,>
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from tensorboard<2.19,>=2.18->tensorflow) (3.
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.11/dist-packages (from werkzeug>=1.0.1->tensorboard<2.19,>=2.
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.11/dist-packages (from rich->keras>=3.5.0->tensorflow) (3
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /usr/local/lib/python3.11/dist-packages (from rich->keras>=3.5.0->tensorflow)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.11/dist-packages (from markdown-it-py>=2.2.0->rich->keras>=3.5.0->te
```

```
import pandas as pd

# Load datasets
insta_df = pd.read_csv("Instagram_sentiment.csv")
twitter_df = pd.read_csv("Twitter_sentiment.csv")

# Add source column
insta_df['platform'] = 'Instagram'
twitter_df['platform'] = 'Twitter'

# Combine datasets
combined_df = pd.concat([insta_df, twitter_df], ignore_index=True)

# Save to a new file
combined_df.to_csv("Combined_SocialMedia_Sentiment.csv", index=False)
```

Install necessary libraries

```
# -*- coding: utf-8 -*-
"""
Fine-tuning_DistilBERT_Social_Media_Sentiment.ipynb

Automatically generated by Colaboratory.

Original file is located at
```

```
    https://colab.research.google.com/notebooks/intro.ipynb

## Fine-tuning a DistilBERT Model for Social Media Sentiment Analysis

This notebook demonstrates how to fine-tune a pre-trained DistilBERT model
from the Hugging Face library for sentiment analysis on a custom dataset
of social media posts (Instagram & Twitter).

**Dataset:** Combined_SocialMedia_Sentiment.csv
**Columns:** (See output of df.columns.tolist() below for actual columns)
**Goal:** Train a model to predict the sentiment score (1-5) based on the text.
"""

# @title Install necessary libraries
# Install transformers, datasets (for handling data), evaluate (for metrics)
!pip install transformers datasets evaluate accelerate -q # -q for quiet installation

# @title Import Libraries
import pandas as pd
import numpy as np
import torch
from sklearn.model_selection import train_test_split
# Corrected import: Removed load_metric as it's deprecated
from datasets import Dataset, DatasetDict
import evaluate # Preferred way for metrics
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer,
    DataCollatorWithPadding,
)
import re # For basic text cleaning
import warnings

# Ignore warnings for cleaner output
warnings.filterwarnings("ignore")

# @title Configuration
# --- Parameters You Might Want to Tune ---
CSV_FILE_PATH = "/content/Combined_SocialMedia_Sentiment.csv"  # <<< IMPORTANT: Upload your CSV file to Colab or change path
# --- VITAL: Updated based on user's df.columns output ---
TEXT_COLUMN = "comment"
# --- VITAL: Updated based on user's df.columns output ---
LABEL_COLUMN = "sentiment"
MODEL_CHECKPOINT = "distilbert-base-uncased" # A good balance of speed and performance
NUM_LABELS = 5  # We have 5 sentiment scores (1 to 5)
TEST_SIZE = 0.2 # Proportion of data to use for validation/testing
RANDOM_STATE = 42 # For reproducibility
BATCH_SIZE = 16 # Adjust based on GPU memory in Colab
LEARNING_RATE = 2e-5
NUM_EPOCHS = 3 # Number of training epochs
OUTPUT_DIR = "sentiment_model_finetuned" # Directory to save the model

# @title Load and Prepare Data
# Load the dataset
try:
    df = pd.read_csv(CSV_FILE_PATH)
    print(f"Successfully loaded {len(df)} rows from {CSV_FILE_PATH}")
    # --- DEBUGGING: Print the actual columns found in the CSV ---
    print("DataFrame Columns:", df.columns.tolist())
    # --- Check this output and update TEXT_COLUMN/LABEL_COLUMN above if needed ---

except FileNotFoundError:
    print(f"ERROR: File not found at {CSV_FILE_PATH}.")
    print("Please upload the file to your Colab environment or update the path.")
    # Exit or raise error if file not found
    raise SystemExit("CSV file not found.")
except Exception as e:
    print(f"An error occurred while loading the CSV: {e}")
    raise SystemExit("Error loading CSV.")


# --- Data Cleaning & Preprocessing ---
print("\n--- Data Preprocessing ---")

# Handle missing text data
```

```
print(f"Initial rows: {len(df)}")
# Check if TEXT_COLUMN exists before trying to dropna
if TEXT_COLUMN not in df.columns:
    print(f"\nERROR: The column specified by TEXT_COLUMN ('{TEXT_COLUMN}') was not found in the DataFrame.")
    print(f"Available columns are: {df.columns.tolist()}")
    print("Please update the TEXT_COLUMN variable in the 'Configuration' cell and rerun.")
    raise KeyError(f"Column '{TEXT_COLUMN}' not found in DataFrame.")

# Handle potential non-numeric or NaN values in LABEL_COLUMN before conversion
if LABEL_COLUMN not in df.columns:
    print(f"\nERROR: The column specified by LABEL_COLUMN ('{LABEL_COLUMN}') was not found in the DataFrame.")
    print(f"Available columns are: {df.columns.tolist()}")
    print("Please update the LABEL_COLUMN variable in the 'Configuration' cell and rerun.")
    raise KeyError(f"Column '{LABEL_COLUMN}' not found in DataFrame.")

# Convert label column to numeric, coercing errors to NaN, then drop rows with NaN labels
df[LABEL_COLUMN] = pd.to_numeric(df[LABEL_COLUMN], errors='coerce')
initial_rows_before_label_dropna = len(df)
df = df.dropna(subset=[LABEL_COLUMN])
print(f"Rows after dropping NaN in label column '{LABEL_COLUMN}': {len(df)} (dropped {initial_rows_before_label_dropna - len(df)})")

# Drop rows with NaN in the text column
initial_rows_before_text_dropna = len(df)
df = df.dropna(subset=[TEXT_COLUMN])
print(f"Rows after dropping missing text in column '{TEXT_COLUMN}': {len(df)} (dropped {initial_rows_before_text_dropna - len(df)})")


# Basic text cleaning (optional, as BERT tokenizers handle much of this)
def clean_text(text):
    if not isinstance(text, str):
        # Attempt to convert to string, or return empty if fails
        try:
            text = str(text)
        except:
             return ""
    text = re.sub(r"http\S+", "", text)  # Remove URLs
    text = re.sub(r"@[A-Za-z0-9_]+", "", text) # Remove mentions
    text = re.sub(r"#[A-Za-z0-9_]+", "", text) # Remove hashtags (optional, might contain sentiment)
    text = re.sub(r"\s+", " ", text).strip() # Remove extra whitespace
    return text

df['cleaned_text'] = df[TEXT_COLUMN].apply(clean_text)

# Prepare labels: sentiment score is 1-5, models need 0-based labels (0-4)
# Ensure the label column is integer type before subtracting
df['labels'] = df[LABEL_COLUMN].astype(int) - 1
print(f"Label distribution (0-4):\n{df['labels'].value_counts().sort_index()}") # Sort index for clarity

# Check if all expected labels (0-4) are present after conversion
expected_labels = set(range(NUM_LABELS))
actual_labels = set(df['labels'].unique())
if not expected_labels.issubset(actual_labels):
     print(f"\nWARNING: Some expected labels (0-{NUM_LABELS-1}) might be missing after processing.")
     print(f"Expected: {expected_labels}, Found: {actual_labels}")
     # Consider adjusting NUM_LABELS if the data truly doesn't have all 5 scores,
     # but usually it's better to keep it as 5 if the original scale was 1-5.

# Keep only necessary columns
df_processed = df[['cleaned_text', 'labels']].rename(columns={'cleaned_text': 'text'})

# --- Split Data ---
print("\n--- Splitting Data ---")
# Check if there's enough data to stratify
if df_processed['labels'].nunique() > 1 and len(df_processed) > 10: # Basic check
    try:
        train_df, test_df = train_test_split(
            df_processed,
            test_size=TEST_SIZE,
            random_state=RANDOM_STATE,
            stratify=df_processed['labels'] # Ensure similar label distribution in splits
        )
    except ValueError as e:
        print(f"Stratify failed: {e}. Splitting without stratify.")
        train_df, test_df = train_test_split(
            df_processed,
            test_size=TEST_SIZE,
            random_state=RANDOM_STATE
```

```
        )
elif len(df_processed) > 0:
     print("Not enough data or classes to stratify. Splitting without stratify.")
     train_df, test_df = train_test_split(
            df_processed,
            test_size=TEST_SIZE,
            random_state=RANDOM_STATE
        )
else:
    raise SystemExit("No data left after preprocessing to split.")


print(f"Training set size: {len(train_df)}")
print(f"Test set size: {len(test_df)}")

# Convert pandas DataFrames to Hugging Face Datasets
train_dataset = Dataset.from_pandas(train_df.reset_index(drop=True))
test_dataset = Dataset.from_pandas(test_df.reset_index(drop=True))

# Combine into a DatasetDict (standard format for Hugging Face)
dataset_dict = DatasetDict({
    'train': train_dataset,
    'test': test_dataset
})

print("\nDatasetDict created:")
print(dataset_dict)

# @title Tokenization
print("\n--- Tokenizing Data ---")
# Load tokenizer associated with the chosen model
tokenizer = AutoTokenizer.from_pretrained(MODEL_CHECKPOINT)

# Function to tokenize the text data
def tokenize_function(examples):
    # `truncation=True` ensures sequences longer than model max length are cut.
    # `padding=True` would pad here, but DataCollator is usually preferred for dynamic padding.
    # Handle potential non-string data in the 'text' column just in case
    texts = [str(t) if t is not None else "" for t in examples["text"]]
    return tokenizer(texts, truncation=True, max_length=512) # Using max_length=512 for BERT-like models

# Apply tokenization to the entire dataset_dict
# `batched=True` processes multiple examples at once for speed
tokenized_datasets = dataset_dict.map(tokenize_function, batched=True)

# Remove the original text column as it's no longer needed after tokenization
# Also remove '__index_level_0__' if it exists from pandas conversion
columns_to_remove = ["text"]
if '__index_level_0__' in tokenized_datasets['train'].column_names:
    columns_to_remove.append('__index_level_0__')

# Check if columns actually exist before removing
existing_columns_to_remove = [col for col in columns_to_remove if col in tokenized_datasets['train'].column_names]
if existing_columns_to_remove:
    tokenized_datasets = tokenized_datasets.remove_columns(existing_columns_to_remove)


# Rename 'labels' column to 'label' if necessary (Trainer expects 'label')
if 'labels' in tokenized_datasets['train'].column_names and 'label' not in tokenized_datasets['train'].column_names:
     tokenized_datasets = tokenized_datasets.rename_column("labels", "label")


# Set format to PyTorch tensors
tokenized_datasets.set_format("torch")

print("\nTokenized datasets prepared:")
print(tokenized_datasets)
# Print example only if dataset is not empty
if len(tokenized_datasets['train']) > 0:
    print(f"\nExample tokenized input: {tokenized_datasets['train'][0]}")
else:
    print("\nTraining dataset is empty after processing.")


# Data collator handles dynamic padding (pads sequences to the longest in a batch)
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
print("\nData Collator initialized.")
```

```

# @title Load Model and Define Metrics
print("\n--- Loading Model & Defining Metrics ---")

# Load the pre-trained model for sequence classification
# `num_labels` must match the number of unique sentiment scores (0-4)
model = AutoModelForSequenceClassification.from_pretrained(
    MODEL_CHECKPOINT,
    num_labels=NUM_LABELS,
    # Add id2label and label2id for better inference later (optional but good practice)
    id2label={i: f"LABEL_{i+1}" for i in range(NUM_LABELS)}, # Maps 0 -> LABEL_1, 1 -> LABEL_2 etc.
    label2id={f"LABEL_{i+1}": i for i in range(NUM_LABELS)}
)
print(f"Loaded model: {MODEL_CHECKPOINT} with {NUM_LABELS} labels.")
# print(f"Model label mapping: {model.config.id2label}") # Uncomment to check mapping

# Define evaluation metrics using the 'evaluate' library
accuracy_metric = evaluate.load("accuracy")
f1_metric = evaluate.load("f1")

def compute_metrics(eval_pred):
    """Computes accuracy and F1 score for evaluation predictions."""
    logits, labels = eval_pred
    # Get the predicted class by finding the index with the highest logit value
    predictions = np.argmax(logits, axis=-1)

    # Calculate accuracy
    accuracy = accuracy_metric.compute(predictions=predictions, references=labels)

    # Calculate F1 score (use 'weighted' for multi-class imbalance)
    f1 = f1_metric.compute(predictions=predictions, references=labels, average="weighted")

    return {
        "accuracy": accuracy["accuracy"],
        "f1": f1["f1"],
    }

print("Metrics computation function defined.")

# @title Configure Training Arguments
print("\n--- Configuring Training ---")

# Define training arguments
# These control various aspects of the fine-tuning process
training_args = TrainingArguments(
    output_dir=OUTPUT_DIR,               # Directory to save model checkpoints and logs
    num_train_epochs=NUM_EPOCHS,         # Total number of training epochs
    per_device_train_batch_size=BATCH_SIZE, # Batch size per GPU/CPU for training
    per_device_eval_batch_size=BATCH_SIZE,  # Batch size per GPU/CPU for evaluation
    learning_rate=LEARNING_RATE,         # Learning rate for the optimizer
    weight_decay=0.01,                   # Strength of weight decay regularization
    evaluation_strategy="epoch",         # Evaluate model performance at the end of each epoch
    save_strategy="epoch",               # Save model checkpoint at the end of each epoch
    logging_strategy="epoch",            # Log training metrics at the end of each epoch
    load_best_model_at_end=True,         # Load the best model checkpoint (based on metric) at the end
    metric_for_best_model="f1",          # Metric used to identify the best model (use F1 for potentially imbalanced classes)
    save_total_limit=2,                  # Only keep the best and the latest checkpoint
    push_to_hub=False,                   # Set to True to push model to Hugging Face Hub (requires login)
    report_to="none",                    # Disable reporting to integrations like wandb/tensorboard unless configured
)

print("Training Arguments configured:")
# print(training_args) # Can be verbose, print if needed

# @title Initialize Trainer
print("\n--- Initializing Trainer ---")

# Check if train dataset is empty before initializing Trainer
if len(tokenized_datasets['train']) == 0:
     raise SystemExit("Training dataset is empty. Cannot initialize Trainer. Please check data loading and preprocessing steps.")


# Initialize the Trainer
# This class orchestrates the fine-tuning process
trainer = Trainer(
    model=model,                         # The instantiated Transformers model to be trained
    args=training_args,                  # Training arguments defined above
```

```
    train_dataset=tokenized_datasets["train"], # Training dataset
    eval_dataset=tokenized_datasets["test"],  # Evaluation dataset (using test set here for simplicity, often a separate validation set
    tokenizer=tokenizer,                # Tokenizer for processing data (needed by Trainer)
    data_collator=data_collator,         # Data collator to handle batch padding
    compute_metrics=compute_metrics,     # Function to compute evaluation metrics
)

print("Trainer initialized.")

# @title Start Fine-tuning
print("\n--- Starting Fine-tuning ---")
# Start the training process
try:
    train_result = trainer.train()
    print("\n--- Fine-tuning Completed ---")

    # Save training metrics
    metrics = train_result.metrics
    trainer.log_metrics("train", metrics)
    trainer.save_metrics("train", metrics)

    # Save the fine-tuned model and tokenizer
    trainer.save_model(OUTPUT_DIR) # Saves the best model due to load_best_model_at_end=True
    tokenizer.save_pretrained(OUTPUT_DIR)
    print(f"\nBest fine-tuned model and tokenizer saved to {OUTPUT_DIR}")

except Exception as e:
    print(f"\nAn error occurred during training: {e}")
    # Potentially add more specific error handling (e.g., CUDA out of memory)
    raise SystemExit("Training failed.")


# @title Evaluate the Fine-tuned Model
print("\n--- Evaluating Model on Test Set ---")
# Evaluate the performance on the test set (loads the best model automatically)
eval_results = trainer.evaluate()

print("\n--- Evaluation Results ---")
print(f"Accuracy: {eval_results.get('eval_accuracy'):.4f}")
print(f"F1 Score (Weighted): {eval_results.get('eval_f1'):.4f}")
# print(eval_results) # Print full results if needed

# Save evaluation metrics
trainer.log_metrics("eval", eval_results)
trainer.save_metrics("eval", eval_results)

# @title Example Prediction (Optional)
print("\n--- Example Prediction ---")

# Load the fine-tuned model explicitly (not strictly necessary as trainer holds the best model)
# model_fine_tuned = AutoModelForSequenceClassification.from_pretrained(OUTPUT_DIR)
# tokenizer_fine_tuned = AutoTokenizer.from_pretrained(OUTPUT_DIR)

# Example text
text_to_classify = "This movie was absolutely fantastic, loved the acting!"
# text_to_classify = "The service was slow and the food was cold."
# text_to_classify = "It was an okay experience, nothing special."

# Use the trainer's model and tokenizer for prediction
current_model = trainer.model
current_tokenizer = trainer.tokenizer

# Tokenize the example text
inputs = current_tokenizer(text_to_classify, return_tensors="pt", truncation=True, padding=True, max_length=512)

# Move inputs to the same device as the model (important if using GPU)
if torch.cuda.is_available() and current_model.device.type == 'cuda':
    inputs = {k: v.to(current_model.device) for k, v in inputs.items()}

# Perform prediction
with torch.no_grad(): # Disable gradient calculation for inference
    logits = current_model(**inputs).logits

# Get the predicted class index
predicted_class_id = torch.argmax(logits, dim=-1).item()

# Map back to original sentiment score (0-4 -> 1-5)
```

```
predicted_sentiment_score = predicted_class_id + 1

print(f"Text: '{text_to_classify}'")
print(f"Predicted Sentiment Score (1-5): {predicted_sentiment_score}")

# You can add a mapping for better interpretation:
sentiment_map = {1: "Very Negative", 2: "Negative", 3: "Neutral", 4: "Positive", 5: "Very Positive"}
print(f"Predicted Sentiment Label: {sentiment_map.get(predicted_sentiment_score, 'Unknown')}")

# Example 2
text_to_classify_2 = "The service was slow and the food was cold."
inputs_2 = current_tokenizer(text_to_classify_2, return_tensors="pt", truncation=True, padding=True, max_length=512)
if torch.cuda.is_available() and current_model.device.type == 'cuda':
    inputs_2 = {k: v.to(current_model.device) for k, v in inputs_2.items()}
with torch.no_grad():
    logits_2 = current_model(**inputs_2).logits
predicted_class_id_2 = torch.argmax(logits_2, dim=-1).item()
predicted_sentiment_score_2 = predicted_class_id_2 + 1
print(f"\nText: '{text_to_classify_2}'")
print(f"Predicted Sentiment Score (1-5): {predicted_sentiment_score_2}")
print(f"Predicted Sentiment Label: {sentiment_map.get(predicted_sentiment_score_2, 'Unknown')}")


```

```
Successfully loaded 2000 rows from /content/Combined_SocialMedia_Sentiment.csv
DataFrame Columns: ['url', 'comment_user', 'comment_user_url', 'comment_date', 'comment', 'likes_number', 'replies_number', 'replies',

--- Data Preprocessing ---
Initial rows: 2000
Rows after dropping NaN in label column 'sentiment': 2000 (dropped 0)
Rows after dropping missing text in column 'comment': 1000 (dropped 1000)
Label distribution (0-4):
labels
0    129
1     19
2     57
3     39
4    756
Name: count, dtype: int64

--- Splitting Data ---
Training set size: 800
Test set size: 200

DatasetDict created:
DatasetDict({
    train: Dataset({
        features: ['text', 'labels'],
        num_rows: 800
    })
    test: Dataset({
        features: ['text', 'labels'],
        num_rows: 200
    })
})

--- Tokenizing Data ---
tokenizer_config.json: 100%  48.0/48.0 [00:00<00:00, 2.95kB/s]
config.json: 100%  483/483 [00:00<00:00, 42.7kB/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 5.60MB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 3.22MB/s]
Map: 100%  800/800 [00:00<00:00, 5703.76 examples/s]
Map: 100%  200/200 [00:00<00:00, 2978.51 examples/s]

Tokenized datasets prepared:
DatasetDict({
    train: Dataset({
        features: ['label', 'input_ids', 'attention_mask'],
        num_rows: 800
    })
    test: Dataset({
        features: ['label', 'input_ids', 'attention_mask'],
        num_rows: 200
    })
})

Example tokenized input: {'label': tensor(4), 'input_ids': tensor([  101, 11320,  6843, 11409,  3527,   102]), 'attention_mask': tensc

Data Collator initialized.

--- Loading Model & Defining Metrics ---
Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better per
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to
model.safetensors: 100%  268M/268M [00:02<00:00, 165MB/s]
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Loaded model: distilbert-base-uncased with 5 labels.
Downloading builder script: 100%  4.20k/4.20k [00:00<00:00, 444kB/s]
Downloading builder script: 100%  6.79k/6.79k [00:00<00:00, 819kB/s]
Metrics computation function defined.

--- Configuring Training ---
Training Arguments configured:

--- Initializing Trainer ---
Trainer initialized.

--- Starting Fine-tuning ---
[150/150 11:00, Epoch 3/3]
 Epoch  Training Loss  Validation Loss  Accuracy  F1
```

1	0.923500	0.809751	0.755000	0.649601
2	0.734600	0.777292	0.755000	0.649601
3	0.679500	0.759023	0.755000	0.649601

```
--- Fine-tuning Completed ---
***** train metrics *****
  epoch                    =        3.0
  total_flos               =    40895GF
  train_loss               =     0.7792
  train_runtime            = 0:11:43.29
  train_samples_per_second =      3.413
  train_steps_per_second   =      0.213

Best fine-tuned model and tokenizer saved to sentiment_model_finetuned

--- Evaluating Model on Test Set ---
```

```
[13/13 00:09]
Trainer.tokenizer is now deprecated. You should use Trainer.processing_class instead.

--- Evaluation Results ---
Accuracy: 0.7550
F1 Score (Weighted): 0.6496
***** eval metrics *****
  epoch                    =        3.0
  eval_accuracy            =      0.755
  eval_f1                  =     0.6496
  eval_loss                =     0.8098
  eval_runtime             = 0:00:09.51
  eval_samples_per_second =      21.03
  eval_steps_per_second    =      1.367

--- Example Prediction ---
Text: 'This movie was absolutely fantastic, loved the acting!'
Predicted Sentiment Score (1-5): 5
Predicted Sentiment Label: Very Positive

Text: 'The service was slow and the food was cold.'
Predicted Sentiment Score (1-5): 5
Predicted Sentiment Label: Very Positive
```

Take 2

Cell 1: Install Libraries

```
## Fine-tuning DistilBERT on Amazon Food Reviews

# @title Cell 1: Install Libraries
!pip install transformers datasets evaluate accelerate torch -q # -q for quiet installation
print("Libraries installed.")
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 491.2/491.2 kB 17.1 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84.0/84.0 kB 6.5 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 3.4 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.8/13.8 MB 36.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 39.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 30.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 2.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 4.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 11.5 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 7.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 4.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 41.4 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 5.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 183.9/183.9 kB 8.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 143.5/143.5 kB 9.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 194.8/194.8 kB 11.5 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source
gcsfs 2025.3.2 requires fsspec==2025.3.2, but you have fsspec 2024.12.0 which is incompatible.
Libraries installed.
```

Cell 2: Import Libraries

```
# @title Cell 2: Import Libraries
import pandas as pd
import numpy as np
import torch
from sklearn.model_selection import train_test_split
from datasets import Dataset, DatasetDict
import evaluate # Preferred way for metrics
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer,
    DataCollatorWithPadding,
)
import re # For basic text cleaning
import warnings

# Ignore warnings for cleaner output
warnings.filterwarnings("ignore")
print("Libraries imported.")
```

```
Libraries imported.
```

Cell 3: Configuration

```
# tile Cell 3: Configuration

# --- Data Settings ---
CSV_FILE_PATH = "//content/Reviews.csv"
SUBSET_SIZE = 50000              # <<< Set to None to load all data (might cause memory issues!), or set a number (e.g., 50000)
TEXT_COLUMN = "Text"
LABEL_COLUMN = "Score"

# --- Model Settings ---
MODEL_CHECKPOINT = "distilbert-base-uncased" # A good balance of speed and performance
NUM_LABELS = 5                   # We have 5 sentiment scores (1 to 5)

# --- Training Settings ---
```

```
TEST_SIZE = 0.2                 # Proportion of data to use for validation/testing
RANDOM_STATE = 42               # For reproducibility
BATCH_SIZE = 16                 # Adjust based on GPU memory in Colab (16 or 32 often work)
LEARNING_RATE = 2e-5            # Standard learning rate for fine-tuning BERT-like models
NUM_EPOCHS = 3                 # Start with 1 epoch for large datasets
OUTPUT_DIR = "sentiment_model_amazon_csv_finetuned" # Directory to save the model

print("Configuration set.")
print(f"CSV file: {CSV_FILE_PATH}")
print(f"Loading subset size: {'All' if SUBSET_SIZE is None else SUBSET_SIZE}")
print(f"Text column: {TEXT_COLUMN}")
print(f"Label column: {LABEL_COLUMN}")
print(f"Model: {MODEL_CHECKPOINT}")
```

```
Configuration set.
CSV file: //content/Reviews.csv
Loading subset size: 50000
Text column: Text
Label column: Score
Model: distilbert-base-uncased
```

Cell 4: Load Data from CSV

```
# title Cell 4: Load Data from CSV

try:
    print(f"Loading data from CSV: {CSV_FILE_PATH}...")
    # Use nrows to load only a subset if SUBSET_SIZE is specified
    df = pd.read_csv(CSV_FILE_PATH, nrows=SUBSET_SIZE)
    print(f"Successfully loaded {len(df)} rows.")
    print("Sample rows:")
    print(df.head())

    # --- DEBUGGING: Print the actual columns found in the CSV ---
    print("\nDataFrame Columns:", df.columns.tolist())
    # --- Check this output and update TEXT_COLUMN/LABEL_COLUMN in Cell 3 if needed ---

    print("\nColumn data types:")
    print(df.info())

    # Verify required columns exist
    if TEXT_COLUMN not in df.columns:
        raise ValueError(f"Text column '{TEXT_COLUMN}' not found in CSV.")
    if LABEL_COLUMN not in df.columns:
        raise ValueError(f"Label column '{LABEL_COLUMN}' not found in CSV.")

except FileNotFoundError:
    print(f"\nERROR: File not found at {CSV_FILE_PATH}.")
    print("Please upload the CSV file to your Colab environment or update the path in Cell 3.")
    raise SystemExit("CSV file not found.")
except ValueError as ve:
    print(f"\nERROR: {ve}")
    print(f"Available columns are: {df.columns.tolist() if 'df' in locals() else 'Could not load DataFrame'}")
    print("Please check the TEXT_COLUMN and LABEL_COLUMN variables in Cell 3.")
    raise SystemExit("Required column not found.")
except Exception as e:
    print(f"\nAn error occurred during CSV loading: {e}")
    raise SystemExit("CSV loading failed.")
```

```
Loading data from CSV: //content/Reviews.csv...
Successfully loaded 50000 rows.
Sample rows:
   Id   ProductId          UserId                      ProfileName  \
0   1  B001E4KFG0  A3SGXH7AUHU8GW                       delmartian
1   2  B00813GRG4  A1D87F6ZCVE5NK                           dll pa
2   3  B000LQOCH0   ABXLMWJIXXAIN  Natalia Corres "Natalia Corres"
3   4  B000UA0QIQ  A395BORC6FGVXV                             Karl
4   5  B006K2ZZ7K  A1UQRSCLF8GW1T    Michael D. Bigham "M. Wassir"

   HelpfulnessNumerator  HelpfulnessDenominator  Score        Time  \
0                     1                       1      5  1303862400
1                     0                       0      1  1346976000
2                     1                       1      4  1219017600
3                     3                       3      2  1307923200
4                     0                       0      5  1350777600
```

```
                Summary                                               Text
0  Good Quality Dog Food  I have bought several of the Vitality canned d...
1      Not as Advertised  Product arrived labeled as Jumbo Salted Peanut...
2  "Delight" says it all  This is a confection that has been around a fe...
3         Cough Medicine  If you are looking for the secret ingredient i...
4            Great taffy  Great taffy at a great price.  There was a wid...

DataFrame Columns: ['Id', 'ProductId', 'UserId', 'ProfileName', 'HelpfulnessNumerator', 'HelpfulnessDenominator', 'Score', 'Time', 'Summ

Column data types:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 10 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Id                      50000 non-null  int64
 1   ProductId               50000 non-null  object
 2   UserId                  50000 non-null  object
 3   ProfileName             49995 non-null  object
 4   HelpfulnessNumerator    50000 non-null  int64
 5   HelpfulnessDenominator  50000 non-null  int64
 6   Score                   50000 non-null  int64
 7   Time                    50000 non-null  int64
 8   Summary                 49998 non-null  object
 9   Text                    50000 non-null  object
dtypes: int64(5), object(5)
memory usage: 3.8+ MB
None
```

Cell 5: Preprocess Data

```
# title Cell 5: Preprocess Data

print("\n--- Data Preprocessing ---")
initial_rows = len(df)
print(f"Initial rows loaded: {initial_rows}")

# 1. Handle missing labels
# Convert label column to numeric, coercing errors to NaN, then drop rows with NaN labels
if not pd.api.types.is_numeric_dtype(df[LABEL_COLUMN]):
    print(f"Converting label column '{LABEL_COLUMN}' to numeric...")
    df[LABEL_COLUMN] = pd.to_numeric(df[LABEL_COLUMN], errors='coerce')

rows_before_label_dropna = len(df)
df = df.dropna(subset=[LABEL_COLUMN])
rows_after_label_dropna = len(df)
print(f"Rows after dropping NaN in label column '{LABEL_COLUMN}': {rows_after_label_dropna} (dropped {rows_before_label_dropna - rows_af

# 2. Handle missing text
rows_before_text_dropna = len(df)
df[TEXT_COLUMN] = df[TEXT_COLUMN].astype(str) # Ensure text column is string type
df = df.dropna(subset=[TEXT_COLUMN]) # Drop rows where text is explicitly NaN
df = df[df[TEXT_COLUMN].str.strip() != ''] # Drop rows where text is empty or only whitespace
rows_after_text_dropna = len(df)
print(f"Rows after dropping missing/empty text in column '{TEXT_COLUMN}': {rows_after_text_dropna} (dropped {rows_before_text_dropna - r


# 3. Basic text cleaning (optional)
def clean_text(text):
    # Remove HTML tags (common in Amazon reviews)
    text = re.sub(r'<.*?>', '', text)
    # Remove URLs
    text = re.sub(r"http\S+", "", text)
    # Remove extra whitespace
    text = re.sub(r"\s+", " ", text).strip()
    return text

print("Applying basic text cleaning...")
df['cleaned_text'] = df[TEXT_COLUMN].apply(clean_text)

# 4. Prepare labels: Score is 1-5, models need 0-based labels (0-4)
# Ensure the label column is integer type before subtracting
df['labels'] = df[LABEL_COLUMN].astype(int) - 1
print(f"\nLabel distribution (0-4):")
print(df['labels'].value_counts().sort_index()) # Sort index for clarity

```

```
# Check if all expected labels (0-4) are present after conversion
expected_labels = set(range(NUM_LABELS))
actual_labels = set(df['labels'].unique())
if not expected_labels.issubset(actual_labels):
     print(f"\nWARNING: Some expected labels (0-{NUM_LABELS-1}) might be missing after processing.")
     print(f"Expected: {expected_labels}, Found: {actual_labels}")
     # Consider adjusting NUM_LABELS if the data truly doesn't have all 5 scores,
     # but usually it's better to keep it as 5 if the original scale was 1-5.

# 5. Keep only necessary columns and rename
df_processed = df[['cleaned_text', 'labels']].rename(columns={'cleaned_text': 'text'})
print("\nPreprocessing finished. Final DataFrame sample:")
print(df_processed.head())
print(f"\nTotal rows after preprocessing: {len(df_processed)}")
```

```
--- Data Preprocessing ---
Initial rows loaded: 50000
Rows after dropping NaN in label column 'Score': 50000 (dropped 0)
Rows after dropping missing/empty text in column 'Text': 50000 (dropped 0)
Applying basic text cleaning...

Label distribution (0-4):
labels
0     4721
1     2814
2     4047
3     7288
4    31130
Name: count, dtype: int64

Preprocessing finished. Final DataFrame sample:
                                                text  labels
0  I have bought several of the Vitality canned d...       4
1  Product arrived labeled as Jumbo Salted Peanut...       0
2  This is a confection that has been around a fe...       3
3  If you are looking for the secret ingredient i...       1
4  Great taffy at a great price. There was a wide...       4

Total rows after preprocessing: 50000
```

Cell 6: Split Data

```
# @title Cell 6: Split Data

print("\n--- Splitting Data ---")
# Check if there's enough data to split and stratify
if len(df_processed) < 10:
     raise SystemExit(f"Not enough data ({len(df_processed)} rows) remaining after preprocessing to split.")

if df_processed['labels'].nunique() > 1:
    try:
        train_df, test_df = train_test_split(
            df_processed,
            test_size=TEST_SIZE,
            random_state=RANDOM_STATE,
            stratify=df_processed['labels'] # Ensure similar label distribution in splits
        )
        print("Data split using stratification.")
    except ValueError as e:
        print(f"Stratify failed: {e}. Splitting without stratify.")
        train_df, test_df = train_test_split(
            df_processed,
            test_size=TEST_SIZE,
            random_state=RANDOM_STATE
        )
else:
     print("Only one class label present. Splitting without stratify.")
     train_df, test_df = train_test_split(
            df_processed,
            test_size=TEST_SIZE,
            random_state=RANDOM_STATE
        )

print(f"Training set size: {len(train_df)}")
print(f"Test set size: {len(test_df)}")

# Convert pandas DataFrames to Hugging Face Datasets
```

```
train_dataset = Dataset.from_pandas(train_df.reset_index(drop=True))
test_dataset = Dataset.from_pandas(test_df.reset_index(drop=True))

# Combine into a DatasetDict (standard format for Hugging Face)
dataset_dict = DatasetDict({
    'train': train_dataset,
    'test': test_dataset
})

print("\nDatasetDict created:")
print(dataset_dict)
```

```
--- Splitting Data ---
Data split using stratification.
Training set size: 40000
Test set size: 10000

DatasetDict created:
DatasetDict({
    train: Dataset({
        features: ['text', 'labels'],
        num_rows: 40000
    })
    test: Dataset({
        features: ['text', 'labels'],
        num_rows: 10000
    })
})
```

Cell 7: Tokenization

```
# @title Cell 7: Tokenization

print("\n--- Tokenizing Data ---")
# Load tokenizer associated with the chosen model
try:
    tokenizer = AutoTokenizer.from_pretrained(MODEL_CHECKPOINT)
    print(f"Tokenizer loaded for {MODEL_CHECKPOINT}")
except Exception as e:
    print(f"Error loading tokenizer for {MODEL_CHECKPOINT}: {e}")
    raise SystemExit("Tokenizer loading failed.")


# Function to tokenize the text data
def tokenize_function(examples):
    # `truncation=True` ensures sequences longer than model max length are cut.
    # `padding=False` (default) - padding will be handled by DataCollator.
    # Handle potential non-string data in the 'text' column just in case
    texts = [str(t) if t is not None else "" for t in examples["text"]]
    return tokenizer(texts, truncation=True, max_length=512) # Using max_length=512 for BERT-like models

# Apply tokenization to the entire dataset_dict
print("Applying tokenization (this may take a while for large datasets)...")
# `batched=True` processes multiple examples at once for speed
tokenized_datasets = dataset_dict.map(tokenize_function, batched=True)
print("Tokenization complete.")

# Remove the original text column as it's no longer needed after tokenization
# Also remove '__index_level_0__' if it exists from pandas conversion
columns_to_remove = ["text"]
if '__index_level_0__' in tokenized_datasets['train'].column_names:
    columns_to_remove.append('__index_level_0__')

# Check if columns actually exist before removing
existing_columns_to_remove = [col for col in columns_to_remove if col in tokenized_datasets['train'].column_names]
if existing_columns_to_remove:
    tokenized_datasets = tokenized_datasets.remove_columns(existing_columns_to_remove)
    print(f"Removed columns: {existing_columns_to_remove}")


# Rename 'labels' column to 'label' if necessary (Trainer expects 'label')
if 'labels' in tokenized_datasets['train'].column_names and 'label' not in tokenized_datasets['train'].column_names:
     tokenized_datasets = tokenized_datasets.rename_column("labels", "label")
     print("Renamed 'labels' column to 'label'.")


```

```
# Set format to PyTorch tensors
tokenized_datasets.set_format("torch")

print("\nTokenized datasets prepared:")
print(tokenized_datasets)
# Print example only if dataset is not empty
if len(tokenized_datasets['train']) > 0:
    print(f"\nExample tokenized input: {tokenized_datasets['train'][0]}")
else:
    print("\nTraining dataset is empty after processing.")


# Data collator handles dynamic padding (pads sequences to the longest in a batch)
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
print("\nData Collator initialized.")
```

```
--- Tokenizing Data ---
Tokenizer loaded for distilbert-base-uncased
Applying tokenization (this may take a while for large datasets)...
Map: 100%    40000/40000 [00:12<00:00, 2225.26 examples/s]

Map: 100%    10000/10000 [00:02<00:00, 3574.63 examples/s]
Tokenization complete.
Removed columns: ['text']
Renamed 'labels' column to 'label'.

Tokenized datasets prepared:
DatasetDict({
    train: Dataset({
        features: ['label', 'input_ids', 'attention_mask'],
        num_rows: 40000
    })
    test: Dataset({
        features: ['label', 'input_ids', 'attention_mask'],
        num_rows: 10000
    })
})

Example tokenized input: {'label': tensor(2), 'input_ids': tensor([  101,  1045,  1005,  2222,  2022,  7481,  1010,  1045,  1005,  104
         1037,  2978, 13971,  2043,  2009,  3310,  2000,  2477,  2066,  2980,
         7967,  1012,  1045,  2428,  5959,  2980,  7967,  1010,  2040,  2987,
         1005,  1056,  1029,  1996,  3277,  2003,  1045,  4025,  2000, 13675,
        10696,  2009,  2043,  1045,  1005,  1049,  2061,  6625,  2006,  1996,
         6411,  2044,  1037,  2146,  2154,  1998,  2123,  1005,  1056,  2514,
         2066, 12959,  2039,  6501,  1010,  6809,  1999, 22940,  1010,  5699,
         1010,  1998,  1037,  3543,  1997,  5474,  2000,  2131, 12090,  2980,
        22940,  1012,  2061,  2073,  2515,  2023,  2681,  2033,  1029,  1045,
         2071,  1010,  2005,  2055,  2322, 16653,  2566, 14771,  1010,  2131,
         5364,  3335,  2980,  7967, 23730,  1998, 15653,  2009,  1999,  2300,
         1998,  4392,  2009,  1010,  2003,  2009, 12090,  1029,  2025,  2428,
         1010,  2021,  1045,  2131,  2026,  8081,  1012,  3291,  2003,  1010,
         2017,  2031,  2000,  3684,  1996,  2300,  1010, 10364,  1999,  1996,
         9898,  1010,  4666,  2009,  1010,  1998,  3066,  2007,  1996, 18856,
        24237,  2015,  1998,  4550,  2039,  1012,  2025,  2026,  5440,  2518,
         2000,  2079,  2012, 29359,  9737,  1006,  2079,  2115,  2190,  2000,
         2025,  2228,  1045,  1005,  1049,  3294, 22692,  1007,  1012,  2085,
         2008,  1045,  2031,  1037, 17710,  9496,  2290,  1010,  1045,  2245,
         2023,  2052,  8081,  2673,  1998,  2507,  2033,  2307,  2980, 22940,
         1012,  2092,  1010,  1045,  2079,  2031, 14057,  1010,  4550,  1010,
         3733,  2000,  7374,  2980, 22940,  1010,  2021,  2049,  1012,  1012,
         1012,  2055,  2004,  2204,  2004,  5364,  3335,  1012,  2061,  2748,
         1010,  2009,  2515, 13225,  2026,  8081,  1010,  1998,  2012,  2069,
         3963, 10250,  2015,  1998,  1019,  1043,  5699,  1010,  1045,  2064,
         3066,  2007,  2008,  2004,  2092,  1012,  1045,  1005,  1049,  2469,
         2065,  3816,  1037,  2210,  2455,  1051,  6597,  2659,  6638,  2431,
         1998,  2431,  1999,  2045,  2009,  2052,  2022,  6022,  2488,  1012,
         1045,  1005,  1049, 22741,  2011,  2035,  2122,  4997,  4391,  1997,
         1047, 10268,  1999,  2236,  1010,  1996,  2801,  2369,  1047, 10268,
         2001, 15106,  1010,  2025,  9352,  1996,  2190, 19645,  2800,  1012,
        11082,  2022,  7481,  1010,  2478,  1037, 22715, 23088,  2121,  1010,
         4840, 13435,  1010,  1998,  1037,  2413,  2811,  1027,  2190,  4157,
         2412,  1010,  6065,  7053,  5572,  2015,  1998, 22421,  1997,  2300,
         1027,  2190,  5572,  2412,  1010,  1998,  1996, 17289,  4084,  2000,
         2980, 22940,  2013, 11969,  1027,  2190,  2980, 22940,  2412,  1012,
         1996,  3291,  2003,  1010,  2035,  1997,  2122,  4725,  5478,  2051,
         2000,  2022,  2985,  2006,  9731,  1010,  2242,  2256,  2554,  2061,
         4600, 29303,  2588,  1012,  2007,  2026, 17710,  9496,  2290,  1010,
         1045,  5256,  2039,  1010,  3328,  2091,  5108,  1010,  2031,  1037,
         2995,  2137,  7451,  3528,  1997,  2092, 11625, 21705,  1010,  3769,
         2028,  1999,  1010,  1998,  2031, 28315,  2075,  2980,  2779,  2000,
         2682,  2779,  4157,  1998,  5572,  2015,  1999,  2625,  2084,  2382,
         3823,  1012,  2061,  2065,  2017,  2215,  2307, 21705,  1010,  2191,
         2009,  4426,  1012,  2065,  2017,  2215,  2204, 21705,  2008,  2024,
        14057,  1998,  4550,  1010,  2131,  1037, 17710,  9496,  2290,  1998,
         2896,  2115, 10908,  1037, 15488, 13623,   999,   102]), 'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

```
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])}

Data Collator initialized.
```

Cell 8: Load Model & Define Metrics

```python
# @title Cell 8: Load Model & Define Metrics

print("\n--- Loading Model & Defining Metrics ---")

# Load the pre-trained model for sequence classification
# `num_labels` must match the number of unique sentiment scores (0-4)
try:
    model = AutoModelForSequenceClassification.from_pretrained(
        MODEL_CHECKPOINT,
        num_labels=NUM_LABELS,
        # Add id2label and label2id for better inference later (optional but good practice)
        id2label={i: f"SCORE_{i+1}" for i in range(NUM_LABELS)}, # Maps 0 -> SCORE_1, 1 -> SCORE_2 etc.
        label2id={f"SCORE_{i+1}": i for i in range(NUM_LABELS)}
    )
    print(f"Loaded model: {MODEL_CHECKPOINT} with {NUM_LABELS} labels.")
    # print(f"Model label mapping: {model.config.id2label}") # Uncomment to check mapping
except Exception as e:
    print(f"Error loading model {MODEL_CHECKPOINT}: {e}")
    raise SystemExit("Model loading failed.")


# Define evaluation metrics using the 'evaluate' library
try:
    accuracy_metric = evaluate.load("accuracy")
    f1_metric = evaluate.load("f1")
    print("Loaded evaluation metrics: Accuracy, F1")
except Exception as e:
    print(f"Error loading evaluation metrics: {e}")
    raise SystemExit("Metrics loading failed.")


def compute_metrics(eval_pred):
    """Computes accuracy and F1 score for evaluation predictions."""
    logits, labels = eval_pred
    # Get the predicted class by finding the index with the highest logit value
    predictions = np.argmax(logits, axis=-1)

    # Calculate accuracy
    accuracy = accuracy_metric.compute(predictions=predictions, references=labels)

    # Calculate F1 score (use 'weighted' for multi-class imbalance)
    f1 = f1_metric.compute(predictions=predictions, references=labels, average="weighted")

    return {
        "accuracy": accuracy["accuracy"],
        "f1": f1["f1"],
    }

print("Metrics computation function defined.")
```

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

--- Loading Model & Defining Metrics ---
Loaded model: distilbert-base-uncased with 5 labels.
Loaded evaluation metrics: Accuracy, F1
Metrics computation function defined.
```

Cell 9: Configure Training Arguments

```python
# @title Cell 9: Configure Training Arguments

print("\n--- Configuring Training ---")

# Define training arguments
```

```
# These control various aspects of the fine-tuning process
training_args = TrainingArguments(
    output_dir=OUTPUT_DIR,               # Directory to save model checkpoints and logs
    num_train_epochs=NUM_EPOCHS,         # Total number of training epochs
    per_device_train_batch_size=BATCH_SIZE, # Batch size per GPU/CPU for training
    per_device_eval_batch_size=BATCH_SIZE*2, # Can often use larger batch size for evaluation
    learning_rate=LEARNING_RATE,         # Learning rate for the optimizer
    weight_decay=0.01,                   # Strength of weight decay regularization
    evaluation_strategy="epoch",         # Evaluate model performance at the end of each epoch
    save_strategy="epoch",               # Save model checkpoint at the end of each epoch
    logging_strategy="epoch",            # Log training metrics at the end of each epoch
    load_best_model_at_end=True,         # Load the best model checkpoint (based on metric) at the end
    metric_for_best_model="f1",          # Metric used to identify the best model (use F1 for potentially imbalanced classes)
    save_total_limit=2,                  # Only keep the best and the latest checkpoint
    fp16=torch.cuda.is_available(),      # Use mixed precision training if GPU is available (faster, less memory)
    push_to_hub=False,                   # Set to True to push model to Hugging Face Hub (requires login)
    report_to="none",                    # Disable reporting to integrations like wandb/tensorboard unless configured
    # Add gradient accumulation to simulate larger batch sizes if memory is limited
    # gradient_accumulation_steps=2, # Uncomment and adjust if needed (effective batch size = BATCH_SIZE * accumulation_steps)
)

print("Training Arguments configured.")
# print(training_args) # Can be verbose, print if needed
```

```
--- Configuring Training ---
Training Arguments configured.
```

Cell 10: Initialize Trainer

```
# @title Cell 10: Initialize Trainer

print("\n--- Initializing Trainer ---")

# Check if train dataset is empty before initializing Trainer
if len(tokenized_datasets['train']) == 0:
     raise SystemExit("Training dataset is empty. Cannot initialize Trainer. Please check data loading and preprocessing steps.")


# Initialize the Trainer
# This class orchestrates the fine-tuning process
trainer = Trainer(
    model=model,                         # The instantiated Transformers model to be trained
    args=training_args,                  # Training arguments defined above
    train_dataset=tokenized_datasets["train"], # Training dataset
    eval_dataset=tokenized_datasets["test"],  # Evaluation dataset
    tokenizer=tokenizer,                 # Tokenizer (needed for padding/batching consistency)
    data_collator=data_collator,         # Data collator to handle batch padding
    compute_metrics=compute_metrics,     # Function to compute evaluation metrics
)

print("Trainer initialized.")
```

```
--- Initializing Trainer ---
Trainer initialized.
```

Cell 11: Start Fine-tuning

```
# @title Cell 11: Start Fine-tuning

print("\n--- Starting Fine-tuning ---")
print(f"Training for {NUM_EPOCHS} epochs...")
# Start the training process
try:
    train_result = trainer.train()
    print("\n--- Fine-tuning Completed ---")

    # Save training metrics
    metrics = train_result.metrics
    trainer.log_metrics("train", metrics)
    trainer.save_metrics("train", metrics)

    # Save the fine-tuned model and tokenizer
    trainer.save_model(OUTPUT_DIR) # Saves the best model due to load_best_model_at_end=True
```

```
    tokenizer.save_pretrained(OUTPUT_DIR)
    print(f"\nBest fine-tuned model and tokenizer saved to {OUTPUT_DIR}")

except Exception as e:
    print(f"\nAn error occurred during training: {e}")
    # Potentially add more specific error handling (e.g., CUDA out of memory)
    raise SystemExit("Training failed.")
```

```
--- Starting Fine-tuning ---
Training for 3 epochs...
```

```
[7500/7500 18:05, Epoch 3/3]
```

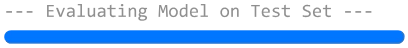
Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.726800	0.650675	0.743900	0.741853
2	0.551200	0.629174	0.763400	0.755655
3	0.440800	0.660513	0.766200	0.760181

```
--- Fine-tuning Completed ---
***** train metrics *****
  epoch                    =        3.0
  total_flos               =  9030560GF
  train_loss               =      0.573
  train_runtime            = 0:18:05.65
  train_samples_per_second =    110.532
  train_steps_per_second   =      6.908

Best fine-tuned model and tokenizer saved to sentiment_model_amazon_csv_finetuned
```

Cell 12: Evaluate Model

```
# @title Cell 12: Evaluate Model

print("\n--- Evaluating Model on Test Set ---")
# Evaluate the performance on the test set (loads the best model automatically)
try:
    # Ensure the OUTPUT_DIR exists where the model was saved by the Trainer
    if not os.path.isdir(OUTPUT_DIR):
         print(f"ERROR: Model output directory '{OUTPUT_DIR}' not found.")
         print("Please ensure training (Cell 11) completed successfully and saved the model.")
         raise SystemExit("Evaluation failed: Model directory not found.")

    eval_results = trainer.evaluate() # Trainer should have loaded the best model

    print("\n--- Evaluation Results ---")
    print(f"Accuracy: {eval_results.get('eval_accuracy', 'N/A'):.4f}")
    print(f"F1 Score (Weighted): {eval_results.get('eval_f1', 'N/A'):.4f}")
    # print(eval_results) # Print full results if needed

    # Save evaluation metrics
    trainer.log_metrics("eval", eval_results)
    trainer.save_metrics("eval", eval_results)
    print("\nEvaluation metrics saved.")

except Exception as e:
    print(f"\nAn error occurred during evaluation: {e}")
    raise SystemExit("Evaluation failed.")
```

```
--- Evaluating Model on Test Set ---
```



```
[313/313 00:25]

--- Evaluation Results ---
Accuracy: 0.7662
F1 Score (Weighted): 0.7602
***** eval metrics *****
  epoch                   =        3.0
  eval_accuracy           =     0.7662
  eval_f1                 =     0.7602
  eval_loss               =     0.6605
  eval_runtime            = 0:00:25.54
  eval_samples_per_second =    391.532
  eval_steps_per_second   =     12.255

Evaluation metrics saved.
```