```
#Author: Pulkit Hooda
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import os, warnings, random
import numpy as np
import torch, transformers, datasets, pyarrow
from pathlib import Path
import pandas as pd
from PIL import Image
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn

#environ setup
os.environ["TOKENIZERS_PARALLELISM"] = "false"
os.environ["WANDB_DISABLED"] = "true"
os.environ["WANDB_SILENT"] = "true"
warnings.filterwarnings("ignore")

#set seed fpr reprod
SEED = 42
def set_seed(seed=SEED):
  random.seed(seed)
  np.random.seed(seed)
  torch.manual_seed(seed)
  torch.cuda.manual_seed_all(seed)
set_seed(SEED)

#paths
DATA_DIR = Path("/content/drive/MyDrive/data/processed_samples")
OUT_DIR = Path("/content/outputs")
OUT_DIR.mkdir(parents=True, exist_ok=True)

#device picker
def pick_device():
  if torch.cuda.is_available(): return "cuda"
  if hasattr(torch.backends, "mps") and torch.backends.mps.is_available(): return "mps"
  return "cpu"

device = pick_device()
use_cuda = device == "cuda"
use_mps = device == "mps"
print("Device:", device)
```

```
Device: cuda
```

```
train_df = pd.read_csv(DATA_DIR / "train_filtered.tsv", sep="\t")
val_df = pd.read_csv(DATA_DIR / "val_filtered.tsv", sep="\t")
test_df = pd.read_csv(DATA_DIR / "test_filtered.tsv", sep="\t")

print("Number of rows in train_df:", len(train_df))
print("Number of rows in val_df:", len(val_df))
print("Number of rows in test_df:", len(test_df))
```

```
Number of rows in train_df: 4777
Number of rows in val_df: 962
Number of rows in test_df: 1901
```

```
def validate_multimodal_df(df: pd.DataFrame) -> pd.DataFrame:
  df = df.copy()
  df = df.loc[:, ~df.columns.str.startswith("Unnamed")]

  #confirm text col
  if 'clean_title' in df.columns:
    df['text'] = df['clean_title'].astype(str)
  elif 'title' in df.columns:
    df['text'] = df['title'].astype(str)
  else:
    raise ValueError(f"No text column found in: {df.columns.tolist()}")

  #confirm label col
  if '2_way_label' in df.columns:
    df['2_way_label'] = pd.to_numeric(df['2_way_label'], errors='coerce')

  #comfirm image path is present
  if 'local_image_path' not in df.columns:
    raise ValueError("Missing 'local_image_path' column for images")

  return df
```

```
#validate all the df
train_df = validate_multimodal_df(train_df)
val_df = validate_multimodal_df(val_df)
test_df = validate_multimodal_df(test_df)
```

```
display(train_df.head())
```

	author	clean_title	created_utc	domain	hasImage	id	image_url	linked_
0	crankbait_XL	feeling lucky	1.353970e+09	NaN	True	c7773wn	http://i.imgur.com/Zu6Sx.jpg	
1	ApiContraption	cutouts	1.397487e+09	NaN	True	cgs3w93	https://31.media.tumblr.com/d7100866f676a6a376...	
2	FerRod05	my ceiling looks like an sd card	1.565055e+09	i.redd.it	True	cmk4af	https://preview.redd.it/zqg8c8fteqe31.jpg?widt...	
3	BlueScreen	join the raaf	1.315071e+09	i.imgur.com	True	k3n1m	https://external-preview.redd.it/q9DNAI6S1OC2v...	
4	vrun1	hangover	1.426882e+09	NaN	True	cplcw5p	http://i.imgur.com/wLCYVSv.jpg	

```
#df only has relevant cols
TEXT_COL, LABEL_COL, IMAGE_COL = "text", "2_way_label", "local_image_path"

for df in (train_df, val_df, test_df):
  df.dropna(subset=[TEXT_COL, LABEL_COL, IMAGE_COL], inplace=True)
  df[LABEL_COL] = df[LABEL_COL].astype(int)

print("Label counts (train):", train_df[LABEL_COL].value_counts())
print("Label counts (val):", val_df[LABEL_COL].value_counts())
print("Label counts (test):", test_df[LABEL_COL].value_counts())
```

```
Label counts (train): 2_way_label
0    2926
1    1851
Name: count, dtype: int64
Label counts (val): 2_way_label
0    597
1    365
Name: count, dtype: int64
Label counts (test): 2_way_label
0    1152
1     749
Name: count, dtype: int64
```

```
from transformers import BertTokenizerFast, CLIPProcessor

#custom dataset class
class MultimodalDataset(Dataset):
  def __init__(self, df, text_tokenizer, image_processor, max_length=128, drive_path="/content/drive/MyDrive/"):
    self.df = df.reset_index(drop=True)
    self.text_tokenizer = text_tokenizer
    self.image_processor = image_processor
    self.max_length = max_length
    self.drive_path = drive_path
    self.data = self._load_data()

  #load and process data
  def _load_data(self):
    processed_data = []
    for idx in range(len(self.df)):
      row = self.df.iloc[idx]
      #process text
      text = str(row[TEXT_COL])
      #encode
      text_encoding = self.text_tokenizer(
        text,
        truncation=True,
        padding="max_length",
        max_length=self.max_length,
        return_tensors="pt"
      )

      #debug
      if idx == 1:
        print(f"Text encoding shape for index {idx}: {text_encoding['input_ids'].shape}")

      #process image
      try:
        #full image path
        image_path = self.drive_path + row[IMAGE_COL]
        #open
        image = Image.open(image_path).convert('RGB')
        #encode
        image_encoding = self.image_processor(
          images=image,
          return_tensors="pt"
        )

        #debug
        if idx == 1:
          print(f"Image encoding shape for index {idx}: {image_encoding['pixel_values'].shape}")

        #pixel encode
        pixel_values = image_encoding['pixel_values'].squeeze(0)

        #debug
        if idx == 1:
          print(f"Pixel values shape for index {idx}: {pixel_values.shape}")

      #never gets here - used for debug prev
      except Exception as e:
        print(f"Error loading image at index {idx} ({image_path}): {e}")
        continue

      #get label
      label_value = row[LABEL_COL]
      labels = None
      try:
        #label tensor
        labels = torch.tensor(int(label_value), dtype=torch.long)

      #never gets here - used for debug prev
      except (ValueError, TypeError) as e:
        print(f"Error processing label for index {idx}: {e}, raw value: {label_value}")
        continue

      #create item and add to processed_data
      item = {
        'input_ids': text_encoding['input_ids'].flatten(),
        'attention_mask': text_encoding['attention_mask'].flatten(),
        'pixel_values': pixel_values,
        'labels': labels
      }
      processed_data.append(item)
    return processed_data


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

  def __getitem__(self, idx):
    return self.data[idx]
```

```
#init tokenizers
text_tokenizer = BertTokenizerFast.from_pretrained("bert-base-uncased")
image_processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
MAX_LEN = 128
```

```
#create datasets
train_dataset = MultimodalDataset(train_df, text_tokenizer, image_processor, MAX_LEN, drive_path="/content/drive/MyD
val_dataset = MultimodalDataset(val_df, text_tokenizer, image_processor, MAX_LEN, drive_path="/content/drive/MyDrive
test_dataset = MultimodalDataset(test_df, text_tokenizer, image_processor, MAX_LEN, drive_path="/content/drive/MyDri

print(f"Dataset sizes: Train={len(train_dataset)}, Val={len(val_dataset)}, Test={len(test_dataset)}")
```

```
tokenizer_config.json: 100%                                         48.0/48.0 [00:00<00:00, 1.18kB/s]

vocab.txt: 100%                                            232k/232k [00:00<00:00, 1.88MB/s]

tokenizer.json: 100%                                         [illegible]

config.json: 100%                                           570/570 [00:00<00:00, 18.8kB/s]
Using a slow image processor as `use_fast` is unset and a slow processor was saved with this model. `use_fast=True` w
preprocessor_config.json: 100%                                         316/316 [00:00<00:00, 8.65kB/s]

tokenizer_config.json: 100%                                         592/592 [00:00<00:00, 48.5kB/s]

config.json:       4.19k/? [00:00<00:00, 141kB/s]

vocab.json:        862k/? [00:00<00:00, 22.4MB/s]

merges.txt:        525k/? [00:00<00:00, 10.2MB/s]

tokenizer.json:        2.22M/? [00:00<00:00, 41.8MB/s]

special_tokens_map.json: 100%                                         389/389 [00:00<00:00, 32.5kB/s]
Text encoding shape for index 1: torch.Size([1, 128])
Image encoding shape for index 1: torch.Size([1, 3, 224, 224])
Pixel values shape for index 1: torch.Size([3, 224, 224])
Text encoding shape for index 1: torch.Size([1, 128])
Image encoding shape for index 1: torch.Size([1, 3, 224, 224])
Pixel values shape for index 1: torch.Size([3, 224, 224])
Text encoding shape for index 1: torch.Size([1, 128])
Image encoding shape for index 1: torch.Size([1, 3, 224, 224])
Pixel values shape for index 1: torch.Size([3, 224, 224])
Dataset sizes: Train=4777, Val=962, Test=1901
```

```
from transformers import BertModel, CLIPModel
import torch.nn.functional as F

#Late Fusion version
class MultimodalBertClipClassifier(nn.Module):

  #dropout_rate=0.1, fusion_dim=512
  def __init__(self, num_labels=2, dropout_rate=0.2, fusion_dim=1024, bert_model_path=None, clip_model_path=None):
    super().__init__()

    #load pretrained models - base model as fall bakc
    if bert_model_path:
      print(f"Loading BERT model from {bert_model_path}")
      self.bert = BertModel.from_pretrained(bert_model_path)
    else:
      print("Loading base BERT model")
      self.bert = BertModel.from_pretrained("bert-base-uncased")

    if clip_model_path:
      print(f"Loading CLIP model from {clip_model_path}")
      if str(clip_model_path).endswith(".pt"):
        self.clip = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
        loaded_state_dict = torch.load(clip_model_path, map_location=device)

        processed_clip_state_dict = {}
        if "model" in loaded_state_dict and "classifier" in loaded_state_dict and isinstance(loaded_state_dict["mode
          print(f"Attempting to extract CLIP weights from 'model' key in {clip_model_path}")
          for k, v in loaded_state_dict["model"].items():
              if k.startswith("model."):
                processed_clip_state_dict[k.replace("model.", "")] = v
              else:
                processed_clip_state_dict[k] = v
        elif any(k.startswith("model.") for k in loaded_state_dict.keys()):
          print(f"Stripping 'model.' prefix from keys in {clip_model_path}")
          for k, v in loaded_state_dict.items():
            processed_clip_state_dict[k.replace("model.", "")] = v
        else:
          processed_clip_state_dict = loaded_state_dict

        self.clip.load_state_dict(processed_clip_state_dict, strict=False)
        print(f"Loaded CLIP state_dict from {clip_model_path} with strict=False")
      else:
        self.clip = CLIPModel.from_pretrained(clip_model_path)
    else:
      print("Loading base CLIP model")
      self.clip = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")


    #get embedding dims should be 768 for BERT
    text_dim = self.bert.config.hidden_size
    print(f"Text embedding dimension: {text_dim}")

    #get embedding dims for clip
    image_dim = self.clip.config.vision_config.projection_dim
    print(f"Image embedding dimension: {image_dim}")

    #late fusion
    self.text_classifier = nn.Sequential(
      nn.Linear(text_dim, fusion_dim // 2),
      nn.ReLU(),
      nn.Dropout(dropout_rate),
      nn.Linear(fusion_dim // 2, num_labels)
    )

    self.image_classifier = nn.Sequential(
      nn.Linear(image_dim, fusion_dim // 2),
      nn.ReLU(),
      nn.Dropout(dropout_rate),
      nn.Linear(fusion_dim // 2, num_labels)
    )

  def forward(self, input_ids, attention_mask, pixel_values, labels=None):
    #text embeddings
    text_outputs = self.bert(
      input_ids=input_ids,
      attention_mask=attention_mask
    )
    text_embeddings = text_outputs.last_hidden_state[:, 0, :]  # [CLS] token

    #image embeddings
    image_outputs = self.clip.get_image_features(pixel_values=pixel_values)

    #separate classifiers
    text_logits = self.text_classifier(text_embeddings)
    image_logits = self.image_classifier(image_outputs)

    #late fusion
    logits = text_logits + image_logits

    return (logits,)

#initialize
model = MultimodalBertClipClassifier(
  num_labels=2,
  bert_model_path="/content/drive/MyDrive/data/models/fine_tuned_bert",
  clip_model_path="/content/drive/MyDrive/data/models/clip_lora_best.pt"
).to(device)

print(f"Model parameters: {sum(p.numel() for p in model.parameters()):,}")
print(f"Trainable parameters: {sum(p.numel() for p in model.parameters() if p.requires_grad):,}")
```

```
Loading BERT model from /content/drive/MyDrive/data/models/fine_tuned_bert
Loading CLIP model from /content/drive/MyDrive/data/models/clip_lora_best.pt
pytorch_model.bin: 100%                                   605M/605M [00:11<00:00, 51.1MB/s]

model.safetensors: 100%                                   605M/605M [00:22<00:00, 33.3MB/s]
Attempting to extract CLIP weights from 'model' key in /content/drive/MyDrive/data/models/clip_lora_best.pt
Loaded CLIP state_dict from /content/drive/MyDrive/data/models/clip_lora_best.pt with strict=False
Text embedding dimension: 768
Image embedding dimension: 512
Model parameters: 261,417,989
Trainable parameters: 261,417,989
```

```
from transformers import BertModel, CLIPModel
import torch.nn.functional as F

#Early Fusion Version
class MultimodalBertClipClassifier(nn.Module):
  #dropout_rate=0.1, fusion_dim=512
  def __init__(self, num_labels=2, dropout_rate=0.2, fusion_dim=1024, bert_model_path=None, clip_model_path=None):
    super().__init__()

    #load pretrained models - base model as fall bakc
    if bert_model_path:
      print(f"Loading BERT model from {bert_model_path}")
      self.bert = BertModel.from_pretrained(bert_model_path)
    else:
      print("Loading base BERT model")
      self.bert = BertModel.from_pretrained("bert-base-uncased")

    if clip_model_path:
      print(f"Loading CLIP model from {clip_model_path}")
      if str(clip_model_path).endswith(".pt"):
        self.clip = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
        loaded_state_dict = torch.load(clip_model_path, map_location=device)
        processed_clip_state_dict = {}
        if "model" in loaded_state_dict and "classifier" in loaded_state_dict and isinstance(loaded_state_dict["mode
          print(f"Attempting to extract CLIP weights from 'model' key in {clip_model_path}")
          for k, v in loaded_state_dict["model"].items():
            if k.startswith("model."):
              processed_clip_state_dict[k.replace("model.", "")] = v
            else:
              processed_clip_state_dict[k] = v
        elif any(k.startswith("model.") for k in loaded_state_dict.keys()):
          print(f"Stripping 'model.' prefix from keys in {clip_model_path}")
          for k, v in loaded_state_dict.items():
            processed_clip_state_dict[k.replace("model.", "")] = v
        else:
          processed_clip_state_dict = loaded_state_dict

        self.clip.load_state_dict(processed_clip_state_dict, strict=False)
        print(f"Loaded CLIP state_dict from {clip_model_path} with strict=False")
      else:
        self.clip = CLIPModel.from_pretrained(clip_model_path)
    else:
      print("Loading base CLIP model")
      self.clip = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")


    #get embedding dims = 768
    text_dim = self.bert.config.hidden_size
    print(f"Text embedding dimension: {text_dim}")
    #image dims
    image_dim = self.clip.config.vision_config.projection_dim
    print(f"Image embedding dimension: {image_dim}")

    #early fusion
    self.fusion_classifier = nn.Sequential(
      nn.Linear(text_dim + image_dim, fusion_dim),
      nn.ReLU(),
      nn.Dropout(dropout_rate),
      nn.Linear(fusion_dim, fusion_dim // 2),
      nn.ReLU(),
      nn.Dropout(dropout_rate),
      nn.Linear(fusion_dim // 2, num_labels)
    )

  def forward(self, input_ids, attention_mask, pixel_values, labels=None): # Add labels=None to the forward signatur
    #text embeddings
    text_outputs = self.bert(
      input_ids=input_ids,
      attention_mask=attention_mask
    )
    text_embeddings = text_outputs.last_hidden_state[:, 0, :]  # [CLS] token

    #image embddings
    image_outputs = self.clip.get_image_features(pixel_values=pixel_values)

    #debug
    #print(f"Text embeddings shape: {text_embeddings.shape}")
    #print(f"Image embeddings shape: {image_outputs.shape}")

    #early fusion
    fused_embeddings = torch.cat([text_embeddings, image_outputs], dim=1)

    #print(f"Fused embeddings shape: {fused_embeddings.shape}")

    #classify
    logits = self.fusion_classifier(fused_embeddings)

    return (logits,)


#initialize
model = MultimodalBertClipClassifier(
  num_labels=2,
  bert_model_path="/content/drive/MyDrive/data/models/fine_tuned_bert",
  clip_model_path="/content/drive/MyDrive/data/models/clip_lora_best.pt"
).to(device)

print(f"Model parameters: {sum(p.numel() for p in model.parameters()):,}")
print(f"Trainable parameters: {sum(p.numel() for p in model.parameters() if p.requires_grad):,}")
```

```
Loading BERT model from /content/drive/MyDrive/data/models/fine_tuned_bert
Loading CLIP model from /content/drive/MyDrive/data/models/clip_lora_best.pt
Attempting to extract CLIP weights from 'model' key in /content/drive/MyDrive/data/models/clip_lora_best.pt
Loaded CLIP state_dict from /content/drive/MyDrive/data/models/clip_lora_best.pt with strict=False
Text embedding dimension: 768
Image embedding dimension: 512
Model parameters: 262,597,123
Trainable parameters: 262,597,123
```

```
from torch.nn.utils.rnn import pad_sequence
class MultimodalDataCollator:
  def __init__(self, return_tensors="pt"):
    self.return_tensors = return_tensors

  def __call__(self, batch):
    #batch size check debug
    #print(f"Processing batch of size: {len(batch)}")

    input_ids = torch.stack([torch.tensor(item['input_ids']) for item in batch])
    attention_mask = torch.stack([torch.tensor(item['attention_mask']) for item in batch])
    pixel_values = torch.stack([item['pixel_values'] for item in batch])

    batch_out = {
      'input_ids': input_ids,
      'attention_mask': attention_mask,
      'pixel_values': pixel_values
    }

    if 'labels' in batch[0]:
      batch_out['labels'] = torch.tensor([item['labels'] for item in batch])
    elif '2_way_label' in batch[0]:
      batch_out['labels'] = torch.tensor([item['2_way_label'] for item in batch])
    else:
      print("No labels in this batch – likely evaluation/prediction batch.")

    return batch_out
data_collator = MultimodalDataCollator(return_tensors="pt")
```

```
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import accuracy_score, f1_score
from transformers import TrainingArguments, Trainer, EarlyStoppingCallback
from torch.utils.data import DataLoader
import numpy as np
from transformers.trainer_utils import EvalPrediction
import torch


#compute class weights
classes = np.array(sorted(train_df[LABEL_COL].unique()))
weights = compute_class_weight(
  class_weight="balanced",
  classes=classes,
  y=train_df[LABEL_COL].values
)
```

```
class_weights = torch.tensor(weights, dtype=torch.float32).to(device)
print("Class weights:", class_weights, "for classes:", classes)

class MultimodalTrainer(Trainer):
  def __init__(self, class_weights=None, **kwargs):
    super().__init__(**kwargs)
    self.class_weights = class_weights

  def compute_loss(self, model, inputs, return_outputs=False, num_items_in_batch=None):
    labels = inputs.pop("labels")
    outputs = model(**inputs)

    #get logits
    logits = outputs[0]

    #handle smaller bathc sizes
    if logits.size(0) != labels.size(0):
      min_size = min(logits.size(0), labels.size(0))
      logits = logits[:min_size]
      labels = labels[:min_size]

    if self.class_weights is not None:
      loss_fct = nn.CrossEntropyLoss(weight=self.class_weights)
    else:
      loss_fct = nn.CrossEntropyLoss()

    loss = loss_fct(logits, labels)
    return (loss, outputs) if return_outputs else loss

  def prediction_step(self, model, inputs, prediction_loss_only, ignore_keys=None):
    labels = inputs.pop("labels") if "labels" in inputs else None

    with torch.no_grad():
      #forward pass
      outputs = model(**inputs)

      #get logits
      if isinstance(outputs, tuple):
        logits = outputs[0]
      else:
        logits = outputs

      #compute loss
      if labels is not None:
        if self.class_weights is not None:
          loss_fct = nn.CrossEntropyLoss(weight=self.class_weights)
        else:
          loss_fct = nn.CrossEntropyLoss()
        loss = loss_fct(logits, labels)
      else:
        loss = None

    #return proper format
    if prediction_loss_only:
      return (loss, None, None)

    return (loss, logits, labels)

def compute_metrics(eval_pred: EvalPrediction):
  #verfy eval_pred
  if not isinstance(eval_pred, EvalPrediction) or not hasattr(eval_pred, 'predictions') or eval_pred.predictions is N
    print(f"Warning: compute_metrics received invalid or empty EvalPrediction object.")
    return {"accuracy": 0.0, "f1": 0.0}

  logits = eval_pred.predictions
  labels = eval_pred.label_ids

  #verify i get tuple
  if isinstance(logits, tuple):
    if len(logits) > 0:
      logits = logits[0]
    else:
      print("Warning: compute_metrics received empty logits tuple from EvalPrediction.")
      return {"accuracy": 0.0, "f1": 0.0}

  #convert to numpy if needed
  if isinstance(logits, torch.Tensor):
    logits = logits.detach().cpu().numpy()
    logits = logits.detach().cpu().numpy()
  if isinstance(labels, torch.Tensor):
    labels = labels.detach().cpu().numpy()

  #predict
  preds = np.argmax(logits, axis=-1)

  return {
    "accuracy": float(accuracy_score(labels, preds)),
    "f1": float(f1_score(labels, preds, average="weighted")),
  }



per_bs = 8 if (use_mps and not use_cuda) else 16
grad_acc = 4 if (use_mps and not use_cuda) else 2

#training args
training_args = TrainingArguments(
  output_dir=f"{OUT_DIR}/multimodal_bert_clip",
  eval_strategy="epoch",
  save_strategy="epoch",
  load_best_model_at_end=True,
  metric_for_best_model="f1",
  greater_is_better=True,

  logging_strategy="steps",
  logging_steps=100,
  per_device_train_batch_size=per_bs,
  per_device_eval_batch_size=max(16, per_bs),
  gradient_accumulation_steps=grad_acc,

  num_train_epochs=8,
  learning_rate=5e-6,
  weight_decay=0.02,
  warmup_ratio=0.10,
  lr_scheduler_type="cosine",

  fp16=use_cuda,
  bf16=False,
  dataloader_num_workers=2,
  report_to="none",
  seed=SEED,
  eval_accumulation_steps=4,
)


#create trainer
trainer = MultimodalTrainer(
  model=model,
  args=training_args,
  train_dataset=train_dataset,
  eval_dataset=val_dataset,
  data_collator=data_collator,
  compute_metrics=compute_metrics,
  class_weights=class_weights,
  callbacks=[EarlyStoppingCallback(early_stopping_patience=3)],
)


#dataset size - debug
print(f"Size of validation dataset before training: {len(val_dataset)}")


#train model
print("Starting multimodal fine-tuning...")
train_result = trainer.train()


#eval validation set
print(f"Size of validation dataset before evaluation: {len(val_dataset)}")
val_metrics = trainer.evaluate(eval_dataset=val_dataset)
print("Validation metrics:", val_metrics)
```

```
Class weights: tensor([0.8163, 1.2904], device='cuda:0') for classes: [0 1]
Size of validation dataset before training: 962
Starting multimodal fine-tuning...
[1050/1200 36:12 < 05:11, 0.48 it/s, Epoch 7/8]
Epoch  Training Loss  Validation Loss  Accuracy  F1
    1       0.652400         0.399856  0.880457  0.881387
    2       0.329700         0.302425  0.877339  0.878179
    3       0.226200         0.288688  0.876299  0.877025
    4       0.136000         0.320961  0.893971  0.893418
    5       0.077800         0.328632  0.888773  0.888228
    6       0.037100         0.380020  0.872141  0.873309
    7       0.020500         0.469259  0.855509  0.855841
Size of validation dataset before evaluation: 962
[61/61 00:02]
```

```
import torch

#test set eval
print("Starting test evaluation...")
test_metrics = trainer.evaluate(eval_dataset=test_dataset)
print("Test metrics:", test_metrics)

'''
output_model_dir = f"{OUT_DIR}/final_model"
Path(output_model_dir).mkdir(parents=True, exist_ok=True)

torch.save(model.state_dict(), f"{output_model_dir}/pytorch_model.bin")

text_tokenizer.save_pretrained(output_model_dir)
image_processor.save_pretrained(output_model_dir)
'''

print("Multimodal fine-tuning and saving done")
```

```
Starting test evaluation...
[61/61 02:11]
Test metrics: {'eval_loss': 0.3120836317539215, 'eval_accuracy': 0.8942661756970016, 'eval_f1': 0.8937028256063102, '
Multimodal fine-tuning and saving done
```