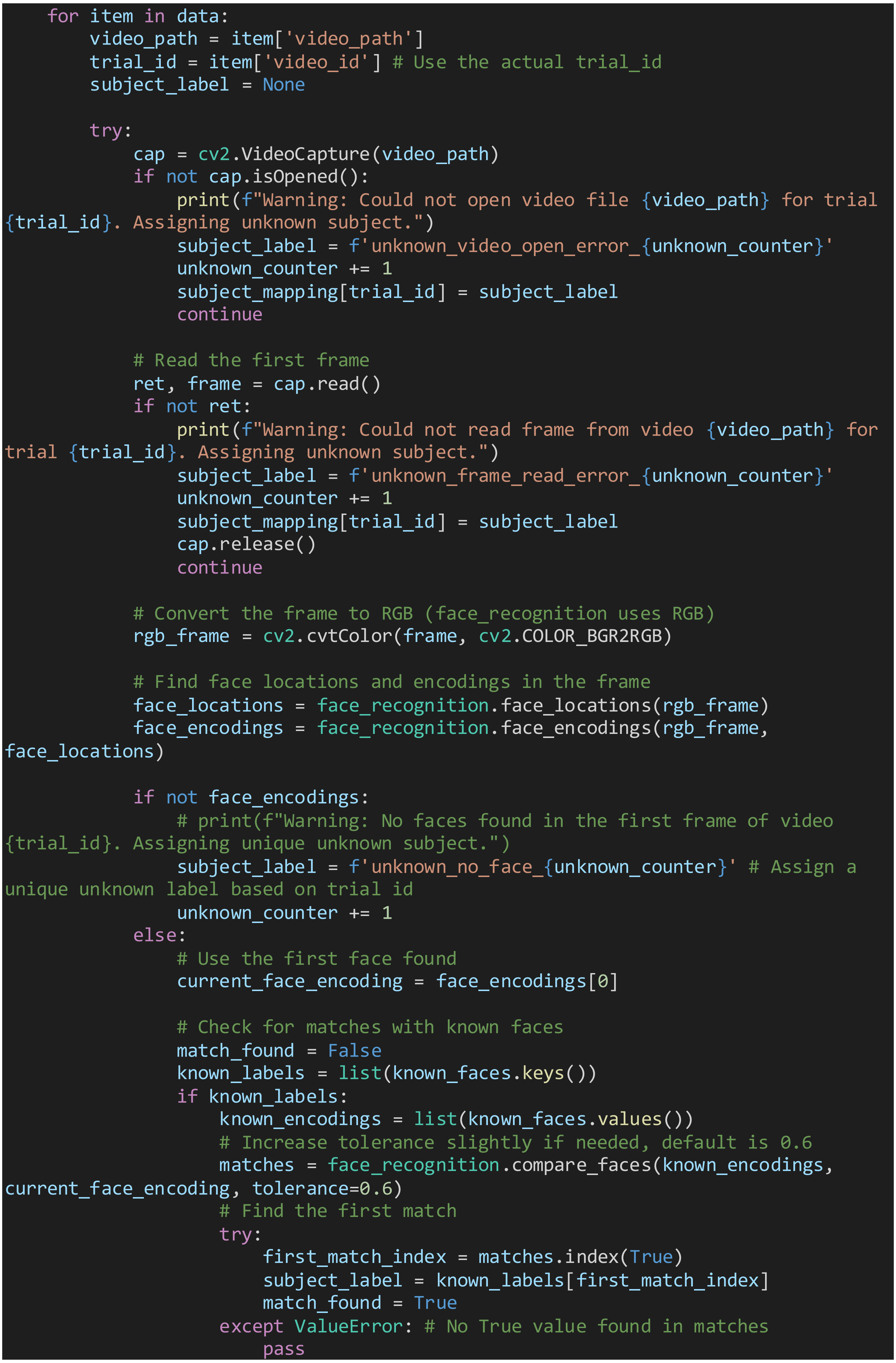
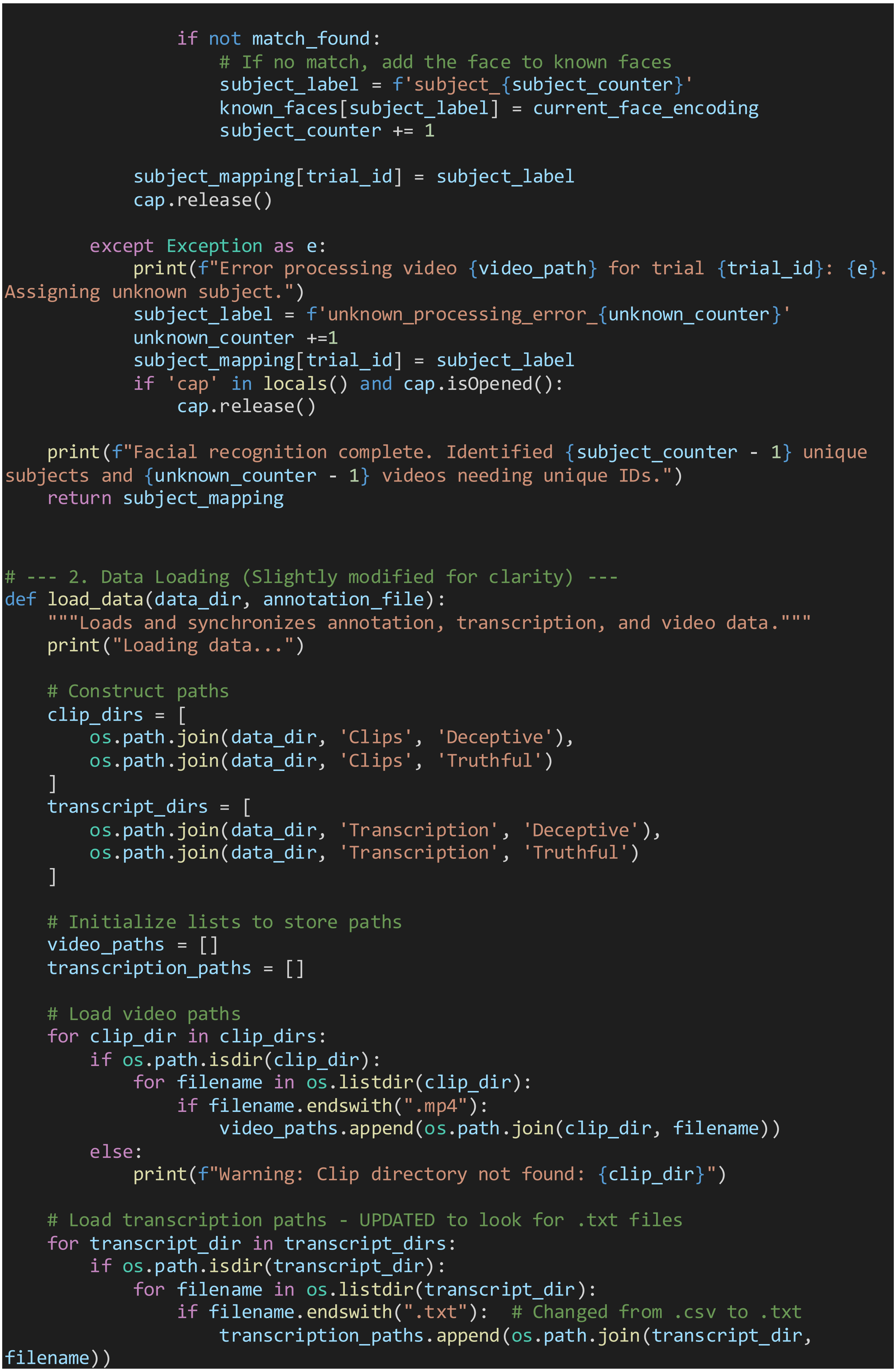
CODE:

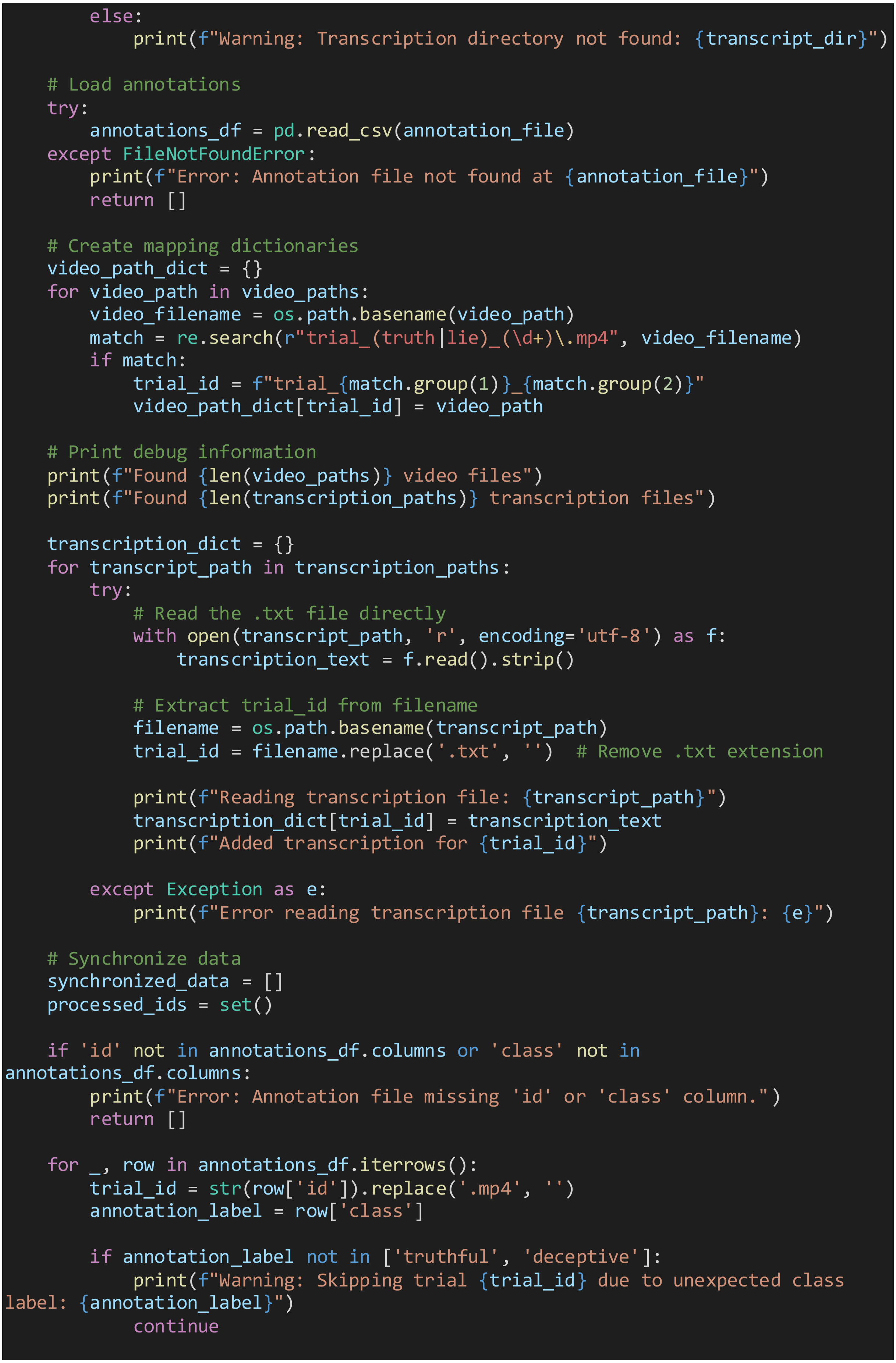
```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from sklearn.model_selection import LeaveOneGroupOut
from transformers import BertTokenizer, BertModel
from sklearn.metrics import accuracy_score, f1_score
import os
import re
import cv2 # OpenCV for video processing
import face_recognition # Now mandatory
import librosa # For audio analysis
from moviepy.editor import VideoFileClip # For extracting audio
import torchvision.models as models
import torchvision.transforms as transforms
from PIL import Image
import time
import warnings

# Suppress warnings from l
# ibraries like moviepy/librosa if needed
warnings.filterwarnings("ignore")

# --- Configuration ---
BERT_MODEL_NAME = 'bert-base-uncased'
VISUAL_MODEL_NAME = 'resnet18' # Using ResNet18 for visual features
AUDIO_N_MFCC = 13 # Number of MFCCs for audio features
VISUAL_FRAMES_TO_SAMPLE = 70 # Number of frames to sample per video for visual
features
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {DEVICE}")

# --- Data Directory Setup ---
BASE_DIR = os.getcwd()  # Get current working directory
data_dir = os.path.join(BASE_DIR, 'Real-life_Deception_Detection_2016')
annotation_file = os.path.join(data_dir, 'Annotation', 'All_Gestures_Deceptive and
Truthful.csv')
# --- 1. Subject Identification (Mandatory Facial Recognition) ---
def identify_subjects_facial_recognition(data):
    """
    Identifies subjects MANDATORILY using facial recognition from the first frame.
    Maps trial_id to a subject label.

    Args:
        data (list): List of dictionaries from load_data, containing 'video_path'
and 'video_id' (trial_id).

    Returns:
        dict: A dictionary mapping trial_ids (video_ids) to subject labels (e.g.,
'subject_1', 'unknown_trial_xyz').
    """
    print("Starting mandatory facial recognition for subject identification...")
    subject_mapping = {}
    known_faces = {}  # Store known face encodings and labels {label: encoding}
    subject_counter = 1
    unknown_counter = 1
```



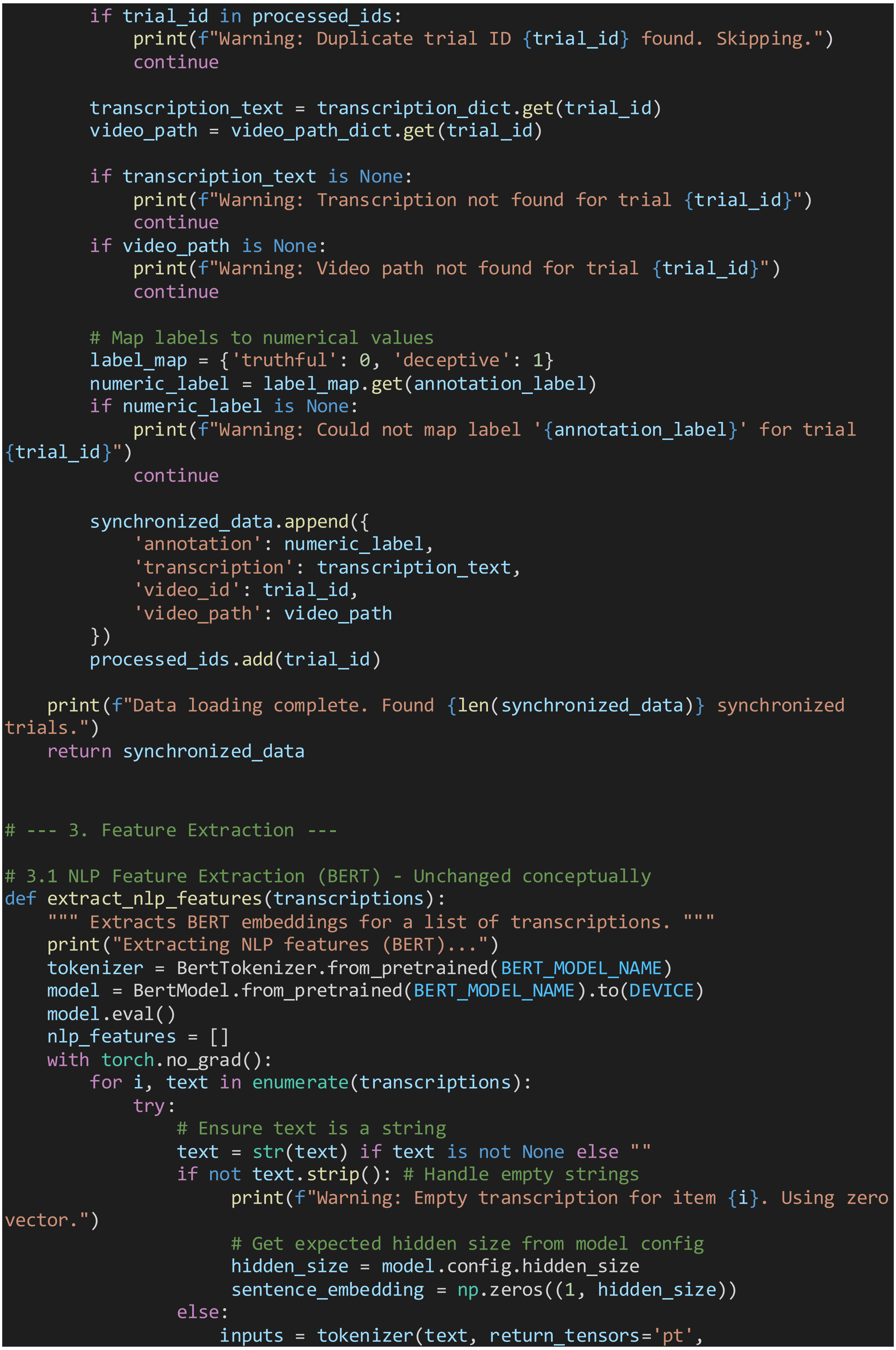
```python
    for item in data:
        video_path = item['video_path']
        trial_id = item['video_id'] # Use the actual trial_id
        subject_label = None

        try:
            cap = cv2.VideoCapture(video_path)
            if not cap.isOpened():
                print(f"Warning: Could not open video file {video_path} for trial {trial_id}. Assigning unknown subject.")
                subject_label = f'unknown_video_open_error_{unknown_counter}'
                unknown_counter += 1
                subject_mapping[trial_id] = subject_label
                continue

            # Read the first frame
            ret, frame = cap.read()
            if not ret:
                print(f"Warning: Could not read frame from video {video_path} for trial {trial_id}. Assigning unknown subject.")
                subject_label = f'unknown_frame_read_error_{unknown_counter}'
                unknown_counter += 1
                subject_mapping[trial_id] = subject_label
                cap.release()
                continue

            # Convert the frame to RGB (face_recognition uses RGB)
            rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

            # Find face locations and encodings in the frame
            face_locations = face_recognition.face_locations(rgb_frame)
            face_encodings = face_recognition.face_encodings(rgb_frame, face_locations)

            if not face_encodings:
                # print(f"Warning: No faces found in the first frame of video {trial_id}. Assigning unique unknown subject.")
                subject_label = f'unknown_no_face_{unknown_counter}' # Assign a unique unknown label based on trial id
                unknown_counter += 1
            else:
                # Use the first face found
                current_face_encoding = face_encodings[0]

                # Check for matches with known faces
                match_found = False
                known_labels = list(known_faces.keys())
                if known_labels:
                    known_encodings = list(known_faces.values())
                    # Increase tolerance slightly if needed, default is 0.6
                    matches = face_recognition.compare_faces(known_encodings, current_face_encoding, tolerance=0.6)
                    # Find the first match
                    try:
                        first_match_index = matches.index(True)
                        subject_label = known_labels[first_match_index]
                        match_found = True
                    except ValueError: # No True value found in matches
                        pass
```



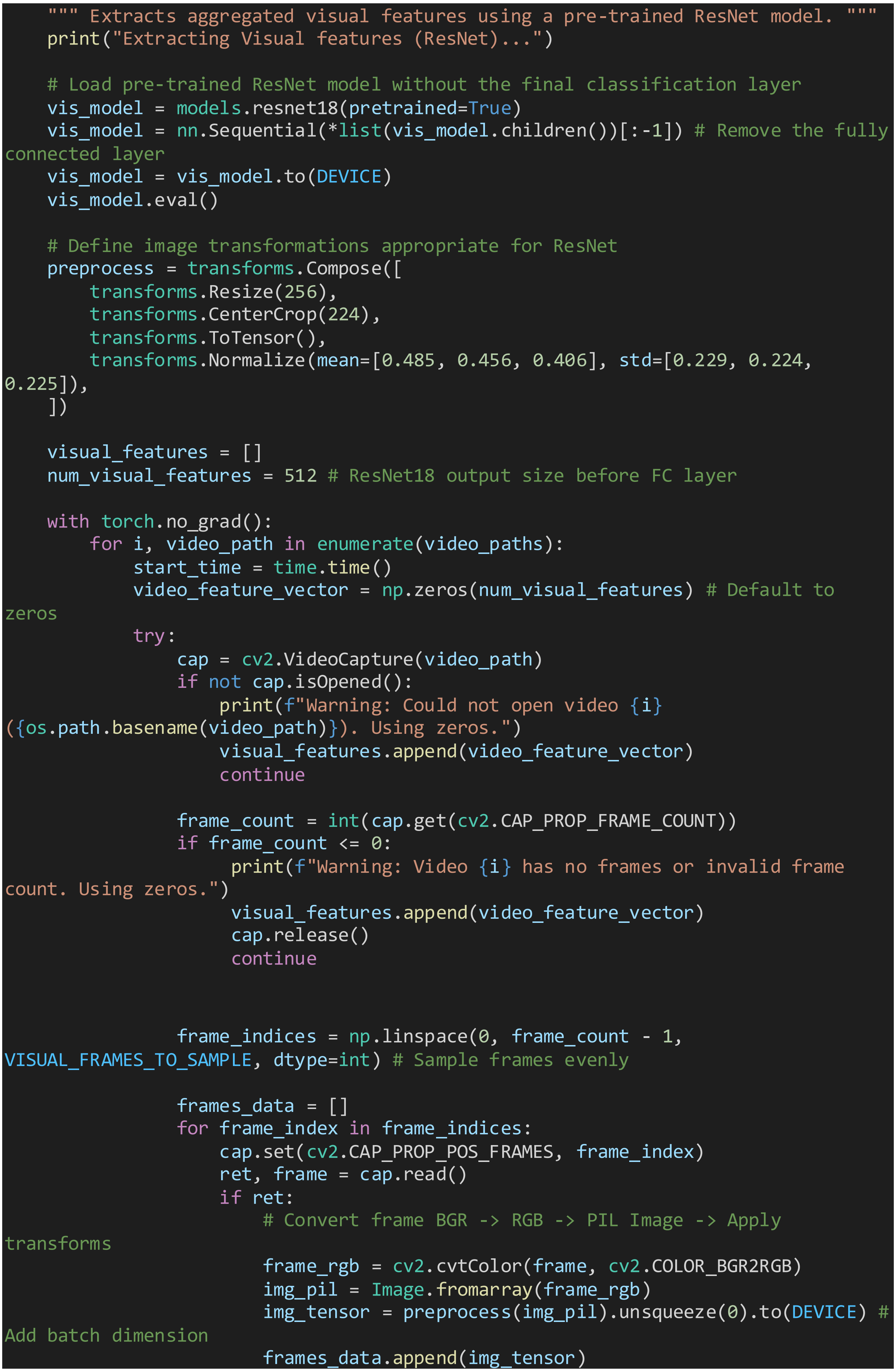
```
                if not match_found:
                    # If no match, add the face to known faces
                    subject_label = f'subject_{subject_counter}'
                    known_faces[subject_label] = current_face_encoding
                    subject_counter += 1

            subject_mapping[trial_id] = subject_label
            cap.release()

        except Exception as e:
            print(f"Error processing video {video_path} for trial {trial_id}: {e}.
Assigning unknown subject.")
            subject_label = f'unknown_processing_error_{unknown_counter}'
            unknown_counter +=1
            subject_mapping[trial_id] = subject_label
            if 'cap' in locals() and cap.isOpened():
                cap.release()

    print(f"Facial recognition complete. Identified {subject_counter - 1} unique
subjects and {unknown_counter - 1} videos needing unique IDs.")
    return subject_mapping


# --- 2. Data Loading (Slightly modified for clarity) ---
def load_data(data_dir, annotation_file):
    """Loads and synchronizes annotation, transcription, and video data."""
    print("Loading data...")

    # Construct paths
    clip_dirs = [
        os.path.join(data_dir, 'Clips', 'Deceptive'),
        os.path.join(data_dir, 'Clips', 'Truthful')
    ]
    transcript_dirs = [
        os.path.join(data_dir, 'Transcription', 'Deceptive'),
        os.path.join(data_dir, 'Transcription', 'Truthful')
    ]

    # Initialize lists to store paths
    video_paths = []
    transcription_paths = []

    # Load video paths
    for clip_dir in clip_dirs:
        if os.path.isdir(clip_dir):
            for filename in os.listdir(clip_dir):
                if filename.endswith(".mp4"):
                    video_paths.append(os.path.join(clip_dir, filename))
        else:
            print(f"Warning: Clip directory not found: {clip_dir}")

    # Load transcription paths - UPDATED to look for .txt files
    for transcript_dir in transcript_dirs:
        if os.path.isdir(transcript_dir):
            for filename in os.listdir(transcript_dir):
                if filename.endswith(".txt"):  # Changed from .csv to .txt
                    transcription_paths.append(os.path.join(transcript_dir,
filename))
```



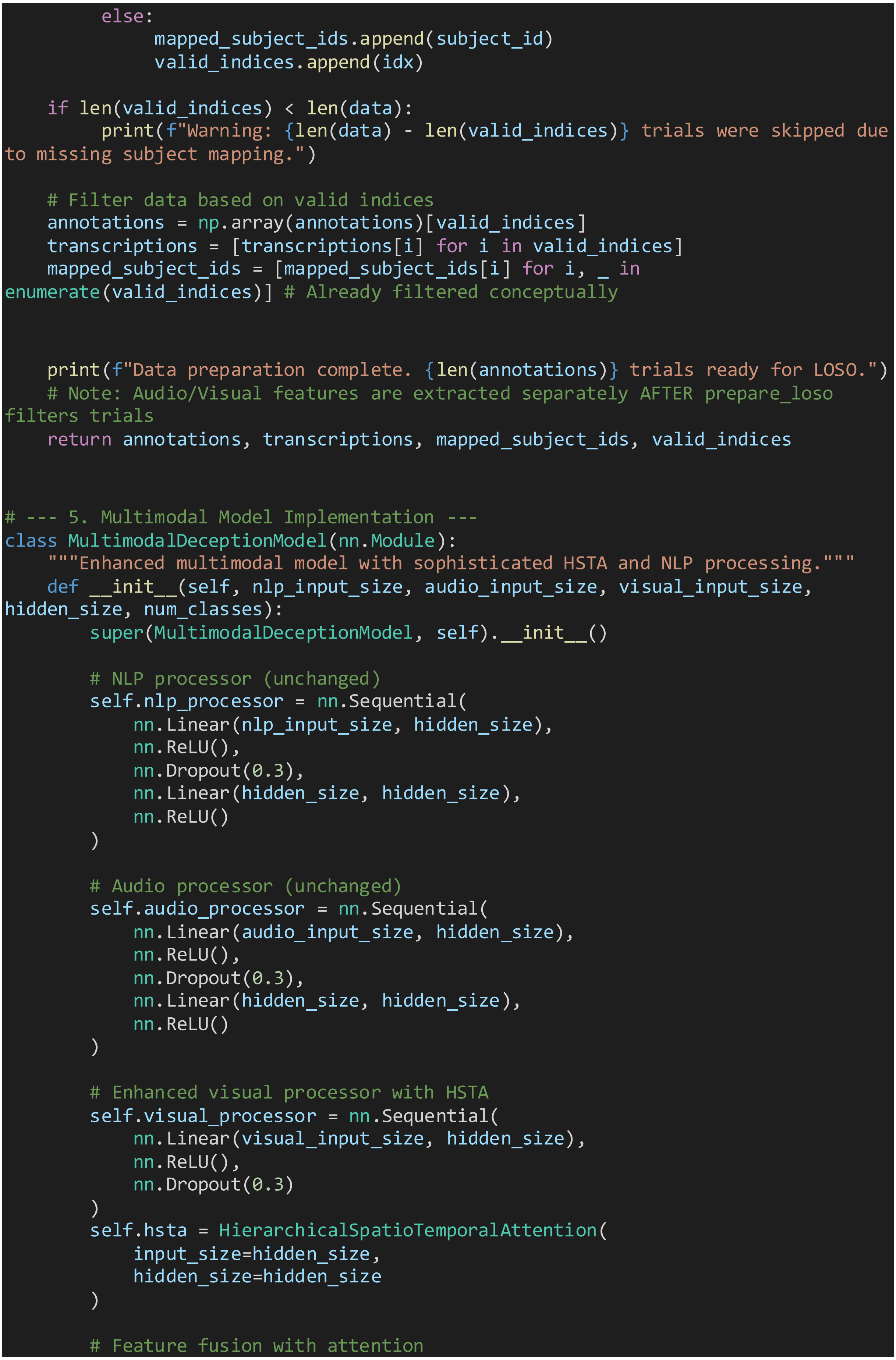
```
        else:
            print(f"Warning: Transcription directory not found: {transcript_dir}")

    # Load annotations
    try:
        annotations_df = pd.read_csv(annotation_file)
    except FileNotFoundError:
        print(f"Error: Annotation file not found at {annotation_file}")
        return []

    # Create mapping dictionaries
    video_path_dict = {}
    for video_path in video_paths:
        video_filename = os.path.basename(video_path)
        match = re.search(r"trial_(truth|lie)_(\d+)\.mp4", video_filename)
        if match:
            trial_id = f"trial_{match.group(1)}_{match.group(2)}"
            video_path_dict[trial_id] = video_path

    # Print debug information
    print(f"Found {len(video_paths)} video files")
    print(f"Found {len(transcription_paths)} transcription files")

    transcription_dict = {}
    for transcript_path in transcription_paths:
        try:
            # Read the .txt file directly
            with open(transcript_path, 'r', encoding='utf-8') as f:
                transcription_text = f.read().strip()

            # Extract trial_id from filename
            filename = os.path.basename(transcript_path)
            trial_id = filename.replace('.txt', '')  # Remove .txt extension

            print(f"Reading transcription file: {transcript_path}")
            transcription_dict[trial_id] = transcription_text
            print(f"Added transcription for {trial_id}")

        except Exception as e:
            print(f"Error reading transcription file {transcript_path}: {e}")

    # Synchronize data
    synchronized_data = []
    processed_ids = set()

    if 'id' not in annotations_df.columns or 'class' not in
annotations_df.columns:
        print(f"Error: Annotation file missing 'id' or 'class' column.")
        return []

    for _, row in annotations_df.iterrows():
        trial_id = str(row['id']).replace('.mp4', '')
        annotation_label = row['class']

        if annotation_label not in ['truthful', 'deceptive']:
            print(f"Warning: Skipping trial {trial_id} due to unexpected class
label: {annotation_label}")
            continue
```



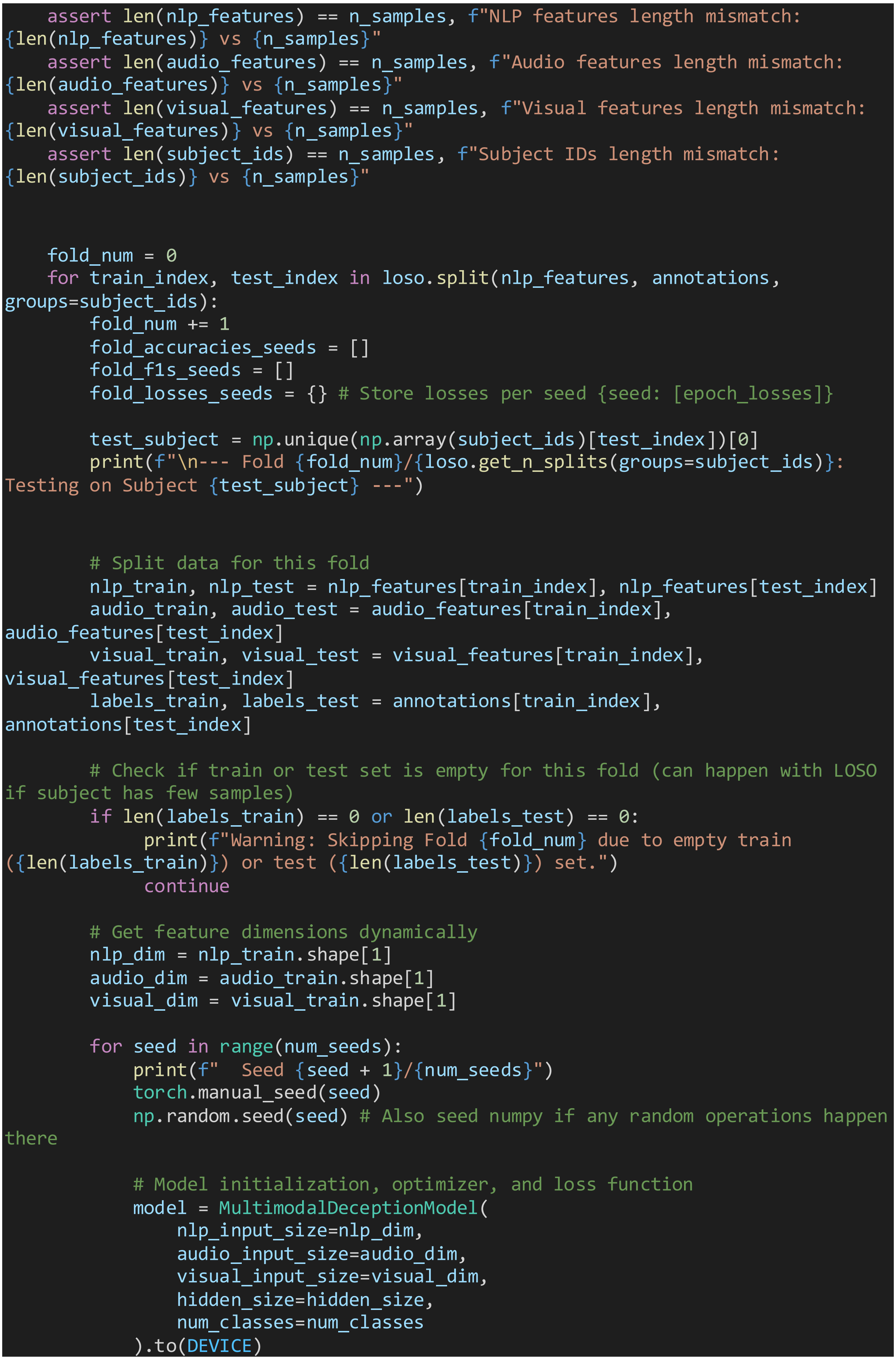
```
        if trial_id in processed_ids:
            print(f"Warning: Duplicate trial ID {trial_id} found. Skipping.")
            continue

        transcription_text = transcription_dict.get(trial_id)
        video_path = video_path_dict.get(trial_id)

        if transcription_text is None:
            print(f"Warning: Transcription not found for trial {trial_id}")
            continue
        if video_path is None:
            print(f"Warning: Video path not found for trial {trial_id}")
            continue

        # Map labels to numerical values
        label_map = {'truthful': 0, 'deceptive': 1}
        numeric_label = label_map.get(annotation_label)
        if numeric_label is None:
            print(f"Warning: Could not map label '{annotation_label}' for trial
{trial_id}")
            continue

        synchronized_data.append({
            'annotation': numeric_label,
            'transcription': transcription_text,
            'video_id': trial_id,
            'video_path': video_path
        })
        processed_ids.add(trial_id)

    print(f"Data loading complete. Found {len(synchronized_data)} synchronized
trials.")
    return synchronized_data


# --- 3. Feature Extraction ---

# 3.1 NLP Feature Extraction (BERT) - Unchanged conceptually
def extract_nlp_features(transcriptions):
    """ Extracts BERT embeddings for a list of transcriptions. """
    print("Extracting NLP features (BERT)...")
    tokenizer = BertTokenizer.from_pretrained(BERT_MODEL_NAME)
    model = BertModel.from_pretrained(BERT_MODEL_NAME).to(DEVICE)
    model.eval()
    nlp_features = []
    with torch.no_grad():
        for i, text in enumerate(transcriptions):
            try:
                # Ensure text is a string
                text = str(text) if text is not None else ""
                if not text.strip(): # Handle empty strings
                     print(f"Warning: Empty transcription for item {i}. Using zero
vector.")
                     # Get expected hidden size from model config
                     hidden_size = model.config.hidden_size
                     sentence_embedding = np.zeros((1, hidden_size))
                else:
                    inputs = tokenizer(text, return_tensors='pt',
                                    truncation=True, padding=True,
max_length=512).to(DEVICE) # Added max_length
                    outputs = model(**inputs)
                    # Mean of the last hidden state
                    sentence_embedding =
outputs.last_hidden_state.mean(dim=1).cpu().numpy() # (1, hidden_size)
                nlp_features.append(sentence_embedding)
            except Exception as e:
                 print(f"Error extracting NLP features for item {i}: {e}. Using
zero vector.")
                 hidden_size = model.config.hidden_size
                 nlp_features.append(np.zeros((1, hidden_size)))


    print("NLP feature extraction complete.")
    # Ensure all features are arrays and handle potential shape issues before
stacking
    processed_features = []
    target_shape = None
    for feat in nlp_features:
        if isinstance(feat, np.ndarray):
             if target_shape is None:
                 target_shape = feat.shape
             # If shape mismatch, pad or truncate (or use zeros as done in
exception handling)
             if feat.shape != target_shape:
                 print(f"Warning: NLP feature shape mismatch {feat.shape} vs
{target_shape}. Using zero vector.")
                 processed_features.append(np.zeros(target_shape))
             else:
                processed_features.append(feat)
        else: # Should not happen if exceptions are caught, but as safeguard
             if target_shape is None: # Need a shape defined first
                 raise ValueError("Cannot process non-array NLP feature without a
target shape.")
             print(f"Warning: Non-array NLP feature found. Using zero vector.")
             processed_features.append(np.zeros(target_shape))

    if not processed_features:
        return np.array([]) # Return empty array if no features were processed

    return np.vstack(processed_features) # (num_trials, hidden_size)


# 3.2 Audio Feature Extraction (MFCCs)
def extract_audio_features(video_paths):
    """ Extracts MFCC features from the audio track of video files. """
    print("Extracting Audio features (MFCCs)...")
    audio_features = []
    temp_audio_dir = "temp_audio"
    if not os.path.exists(temp_audio_dir):
        os.makedirs(temp_audio_dir)

    num_features = AUDIO_N_MFCC * 2 # Mean and Std Dev for each MFCC

    for i, video_path in enumerate(video_paths):
        start_time = time.time()
        temp_audio_path = os.path.join(temp_audio_dir, f"temp_{i}.wav")
        feature_vector = np.zeros(num_features) # Default to zeros

        try:
            # Extract audio using moviepy
            with VideoFileClip(video_path) as video_clip:
                 if video_clip.audio is None:
                      print(f"Warning: Video {i} ({os.path.basename(video_path)}) has no audio track. Using zeros.")
                 else:
                    video_clip.audio.write_audiofile(temp_audio_path, codec='pcm_s16le', logger=None) # Use logger=None to reduce console output

            # Load audio and extract MFCCs using librosa
            y, sr = librosa.load(temp_audio_path, sr=None) # Load with native sample rate
            if len(y) > 0: # Check if audio signal is not empty
                mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=AUDIO_N_MFCC)
                mfccs_mean = np.mean(mfccs, axis=1)
                mfccs_std = np.std(mfccs, axis=1)
                feature_vector = np.concatenate((mfccs_mean, mfccs_std))
            else:
                 print(f"Warning: Audio signal empty after extraction for video {i}. Using zeros.")


        except Exception as e:
            print(f"Error extracting audio features for video {i} ({os.path.basename(video_path)}): {e}. Using zeros.")
            # Ensure feature_vector remains zeros

        finally:
            # Clean up temporary audio file
            if os.path.exists(temp_audio_path):
                try:
                    os.remove(temp_audio_path)
                except Exception as e:
                    print(f"Warning: Could not remove temp audio file {temp_audio_path}: {e}")

        audio_features.append(feature_vector)
        # print(f"Processed audio for video {i+1}/{len(video_paths)} in {time.time() - start_time:.2f}s") # Optional progress

    # Clean up temp directory if empty
    try:
        if not os.listdir(temp_audio_dir):
            os.rmdir(temp_audio_dir)
    except Exception as e:
        print(f"Warning: Could not remove temp audio directory {temp_audio_dir}: {e}")


    print("Audio feature extraction complete.")
    return np.array(audio_features) # (num_trials, num_audio_features)


# 3.3 Visual Feature Extraction (ResNet)
def extract_visual_features(video_paths):
```



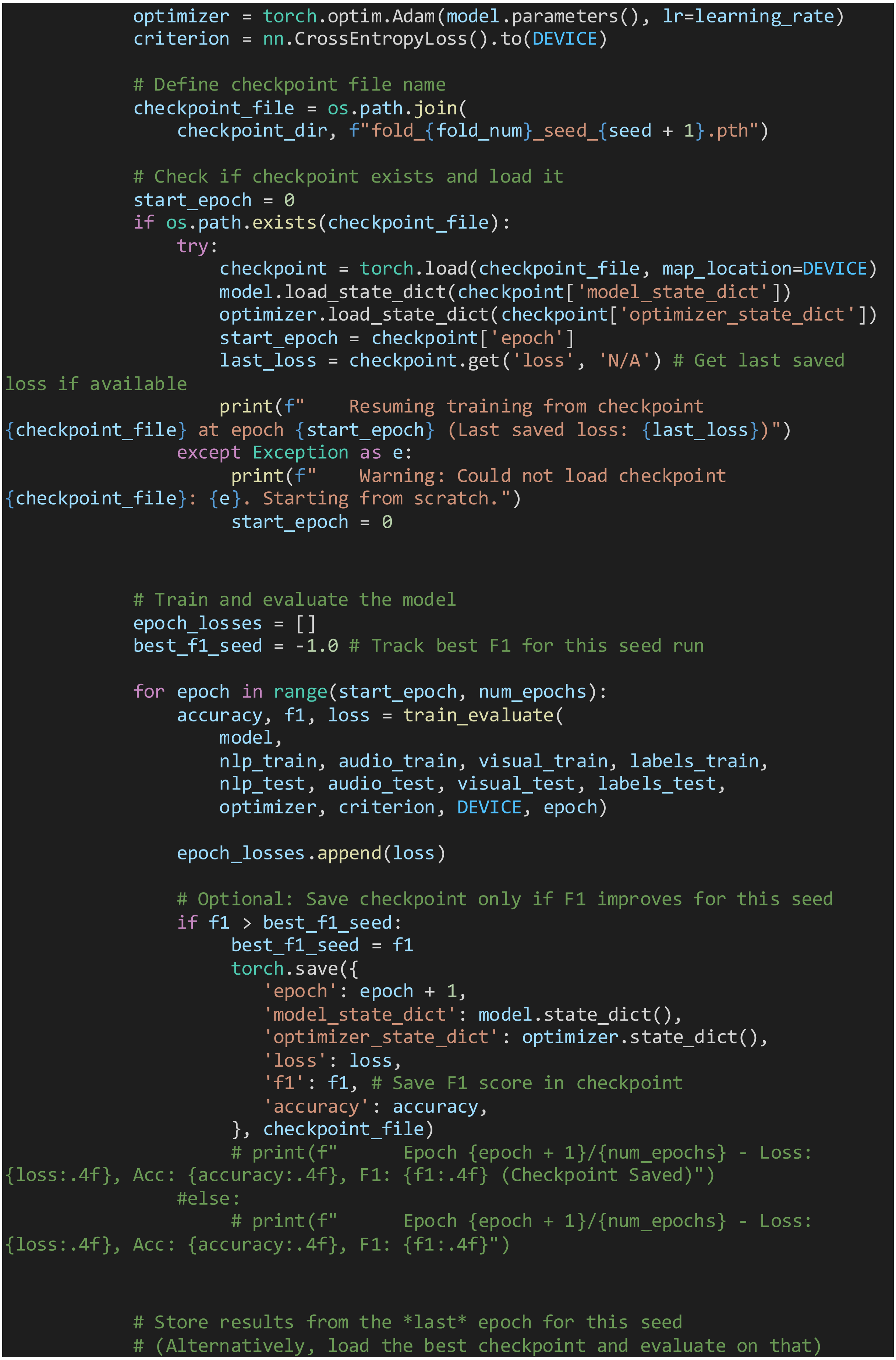
```
    """ Extracts aggregated visual features using a pre-trained ResNet model. """
    print("Extracting Visual features (ResNet)...")

    # Load pre-trained ResNet model without the final classification layer
    vis_model = models.resnet18(pretrained=True)
    vis_model = nn.Sequential(*list(vis_model.children())[:-1]) # Remove the fully
connected layer
    vis_model = vis_model.to(DEVICE)
    vis_model.eval()

    # Define image transformations appropriate for ResNet
    preprocess = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225]),
    ])

    visual_features = []
    num_visual_features = 512 # ResNet18 output size before FC layer

    with torch.no_grad():
        for i, video_path in enumerate(video_paths):
            start_time = time.time()
            video_feature_vector = np.zeros(num_visual_features) # Default to
zeros
            try:
                cap = cv2.VideoCapture(video_path)
                if not cap.isOpened():
                    print(f"Warning: Could not open video {i}
({os.path.basename(video_path)}). Using zeros.")
                    visual_features.append(video_feature_vector)
                    continue

                frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
                if frame_count <= 0:
                     print(f"Warning: Video {i} has no frames or invalid frame
count. Using zeros.")
                     visual_features.append(video_feature_vector)
                     cap.release()
                     continue


                frame_indices = np.linspace(0, frame_count - 1,
VISUAL_FRAMES_TO_SAMPLE, dtype=int) # Sample frames evenly

                frames_data = []
                for frame_index in frame_indices:
                    cap.set(cv2.CAP_PROP_POS_FRAMES, frame_index)
                    ret, frame = cap.read()
                    if ret:
                        # Convert frame BGR -> RGB -> PIL Image -> Apply
transforms
                        frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
                        img_pil = Image.fromarray(frame_rgb)
                        img_tensor = preprocess(img_pil).unsqueeze(0).to(DEVICE) #
Add batch dimension
                        frames_data.append(img_tensor)
                    # else: # Optional: Warn if specific frame fails
                        # print(f"Warning: Could not read frame {frame_index} from video {i}.")


                cap.release()

                if frames_data:
                    # Stack frame tensors and pass through the model
                    batch_tensor = torch.cat(frames_data, dim=0)
                    frame_outputs = vis_model(batch_tensor) # (num_sampled_frames, num_visual_features, 1, 1)
                    frame_outputs = frame_outputs.squeeze() # Remove trailing 1s -> (num_sampled_frames, num_visual_features)
                    # Aggregate features (e.g., mean pooling)
                    video_feature_vector = torch.mean(frame_outputs, dim=0).cpu().numpy()
                else:
                    print(f"Warning: No frames could be processed for video {i}. Using zeros.")
                    # video_feature_vector remains zeros


            except Exception as e:
                print(f"Error extracting visual features for video {i} ({os.path.basename(video_path)}): {e}. Using zeros.")
                if 'cap' in locals() and cap.isOpened():
                    cap.release()
                # video_feature_vector remains zeros

            visual_features.append(video_feature_vector)
            # print(f"Processed visual for video {i+1}/{len(video_paths)} in {time.time() - start_time:.2f}s")

    print("Visual feature extraction complete.")
    return np.array(visual_features) # (num_trials, num_visual_features)



# --- 4. Data Preparation for LOSO ---
def prepare_loso(data, subject_mapping):
    """
    Prepares data for LOSO cross-validation using the mandatory subject mapping.
    Returns raw transcriptions for later NLP processing.
    """
    print("Preparing data for LOSO...")
    annotations = [item['annotation'] for item in data]
    transcriptions = [item['transcription'] for item in data] # Keep raw text
    video_ids = [item['video_id'] for item in data]

    # Map video_ids (trial_ids) to subject IDs using the facial recognition mapping
    mapped_subject_ids = []
    valid_indices = [] # Keep track of trials with successful subject mapping
    for idx, video_id in enumerate(video_ids):
         subject_id = subject_mapping.get(video_id)
         if subject_id is None:
              print(f"Critical Warning: No subject mapping found for video_id {video_id}. This trial will be skipped in prepare_loso.")
```



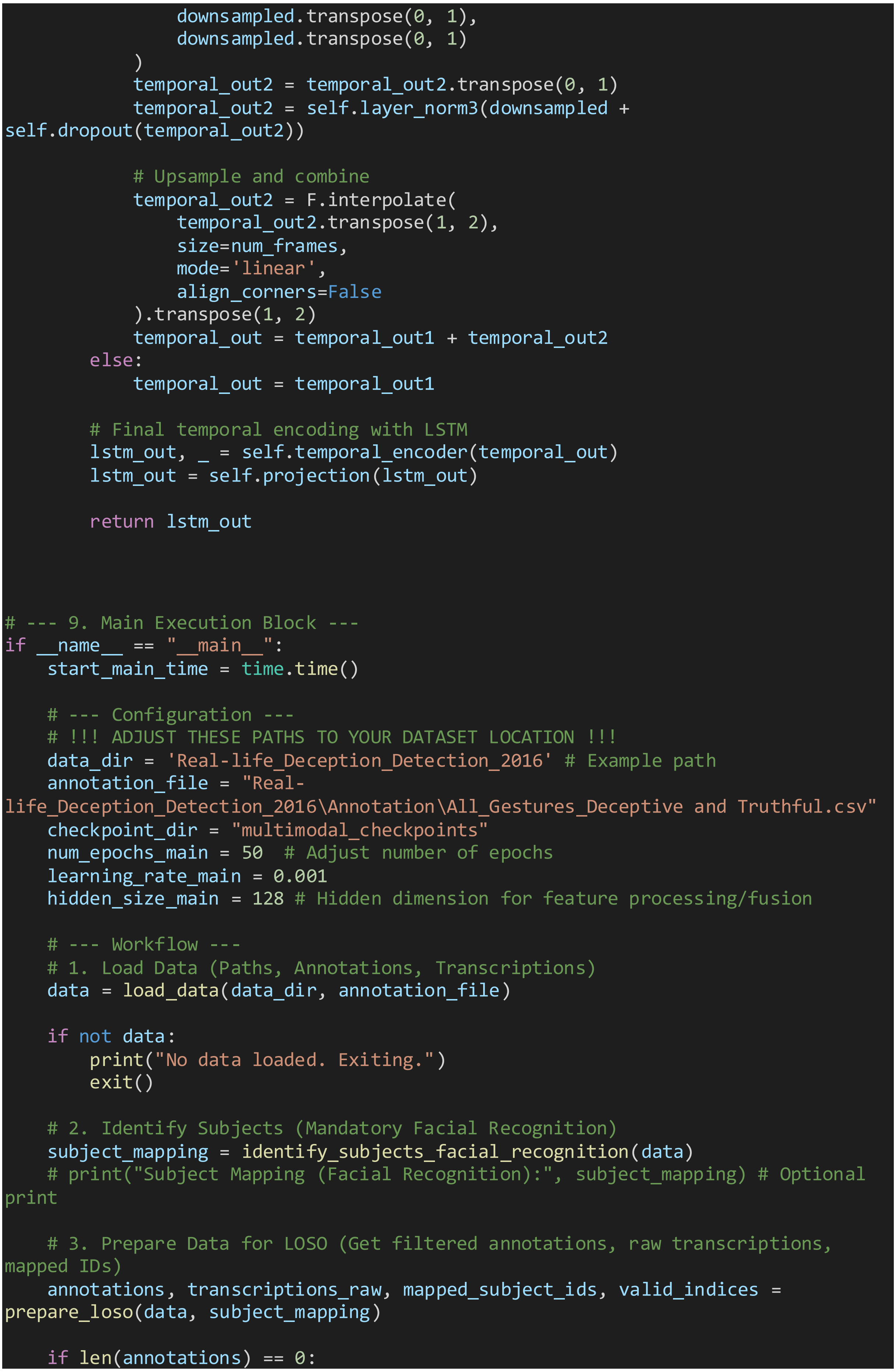
```
         else:
              mapped_subject_ids.append(subject_id)
              valid_indices.append(idx)

    if len(valid_indices) < len(data):
         print(f"Warning: {len(data) - len(valid_indices)} trials were skipped due
to missing subject mapping.")

    # Filter data based on valid indices
    annotations = np.array(annotations)[valid_indices]
    transcriptions = [transcriptions[i] for i in valid_indices]
    mapped_subject_ids = [mapped_subject_ids[i] for i, _ in
enumerate(valid_indices)] # Already filtered conceptually


    print(f"Data preparation complete. {len(annotations)} trials ready for LOSO.")
    # Note: Audio/Visual features are extracted separately AFTER prepare_loso
filters trials
    return annotations, transcriptions, mapped_subject_ids, valid_indices



# --- 5. Multimodal Model Implementation ---
class MultimodalDeceptionModel(nn.Module):
    """Enhanced multimodal model with sophisticated HSTA and NLP processing."""
    def __init__(self, nlp_input_size, audio_input_size, visual_input_size,
hidden_size, num_classes):
        super(MultimodalDeceptionModel, self).__init__()

        # NLP processor (unchanged)
        self.nlp_processor = nn.Sequential(
            nn.Linear(nlp_input_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU()
        )

        # Audio processor (unchanged)
        self.audio_processor = nn.Sequential(
            nn.Linear(audio_input_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU()
        )

        # Enhanced visual processor with HSTA
        self.visual_processor = nn.Sequential(
            nn.Linear(visual_input_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(0.3)
        )
        self.hsta = HierarchicalSpatioTemporalAttention(
            input_size=hidden_size,
            hidden_size=hidden_size
        )

        # Feature fusion with attention
        self.fusion_attention = nn.MultiheadAttention(
            embed_dim=hidden_size * 3,
            num_heads=8,
            dropout=0.1
        )

        # Final classifier
        self.classifier = nn.Sequential(
            nn.Linear(hidden_size * 3, hidden_size),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(hidden_size, num_classes)
        )

    def forward(self, nlp_data, audio_data, visual_data):
        # Process each modality
        nlp_processed = self.nlp_processor(nlp_data)
        audio_processed = self.audio_processor(audio_data)

        # Process visual data with HSTA
        visual_processed = self.visual_processor(visual_data)
        # Reshape for HSTA (batch_size, num_frames, hidden_size)
        batch_size = visual_processed.size(0)
        visual_processed = visual_processed.view(batch_size, -1,
visual_processed.size(-1))
        visual_processed = self.hsta(visual_processed)
        # Take the last frame's representation
        visual_processed = visual_processed[:, -1, :]

        # Concatenate features
        fused_features = torch.cat((nlp_processed, audio_processed,
visual_processed), dim=1)

        # Apply fusion attention
        fused_features = fused_features.unsqueeze(0)  # Add sequence dimension
        fused_features, _ = self.fusion_attention(fused_features, fused_features,
fused_features)
        fused_features = fused_features.squeeze(0)

        # Final classification
        output = self.classifier(fused_features)
        return output

# --- 6. Training and Evaluation (Modified for Multimodal) ---
def train_evaluate(model,
                   nlp_train, audio_train, visual_train, labels_train,
                   nlp_test, audio_test, visual_test, labels_test,
                   optimizer, criterion, device, epoch):
    """ Trains and evaluates the multimodal model for one epoch. """
    model.train()
    optimizer.zero_grad()

    # Move training data to device
    nlp_train_tensor = torch.FloatTensor(nlp_train).to(device)
    audio_train_tensor = torch.FloatTensor(audio_train).to(device)
    visual_train_tensor = torch.FloatTensor(visual_train).to(device)
    labels_train_tensor = torch.LongTensor(labels_train).to(device)

    # Forward pass (Training)
    outputs = model(nlp_train_tensor, audio_train_tensor, visual_train_tensor)
    loss = criterion(outputs, labels_train_tensor)

    # Backward pass and optimization
    loss.backward()
    optimizer.step()

    # Evaluation
    model.eval()
    with torch.no_grad():
        # Move test data to device
        nlp_test_tensor = torch.FloatTensor(nlp_test).to(device)
        audio_test_tensor = torch.FloatTensor(audio_test).to(device)
        visual_test_tensor = torch.FloatTensor(visual_test).to(device)
        labels_test_tensor = torch.LongTensor(labels_test).to(device)

        # Forward pass (Evaluation)
        outputs_test = model(nlp_test_tensor, audio_test_tensor,
visual_test_tensor)
        _, predicted = torch.max(outputs_test.data, 1)

        # Calculate metrics
        labels_test_cpu = labels_test_tensor.cpu().numpy()
        predicted_cpu = predicted.cpu().numpy()
        accuracy = accuracy_score(labels_test_cpu, predicted_cpu)
        f1 = f1_score(labels_test_cpu, predicted_cpu, average='weighted',
zero_division=0) # Added zero_division

    return accuracy, f1, loss.item()


# --- 7. Run LOSO Cross-Validation (Modified for Multimodal) ---
def run_loso(annotations, nlp_features, audio_features, visual_features,
subject_ids,
             checkpoint_dir="checkpoints", num_epochs=50, learning_rate=0.001,
hidden_size=128):
    """ Runs LOSO cross-validation for the multimodal model. """
    print("Starting LOSO Cross-Validation...")
    loso = LeaveOneGroupOut()
    all_accuracies = []
    all_f1s = []
    all_losses = []  # To store loss per fold and seed
    num_seeds = 3 # Keep number of seeds

    # Create checkpoint directory if it doesn't exist
    if not os.path.exists(checkpoint_dir):
        os.makedirs(checkpoint_dir)

    num_classes = len(np.unique(annotations))
    if num_classes < 2:
        print(f"Error: Only {num_classes} unique class found. Cannot perform
classification.")
        return

    # Use subject_ids (mapped) as groups for LOSO
    # Ensure all feature arrays have the same number of samples as
annotations/subject_ids
    n_samples = len(annotations)
```



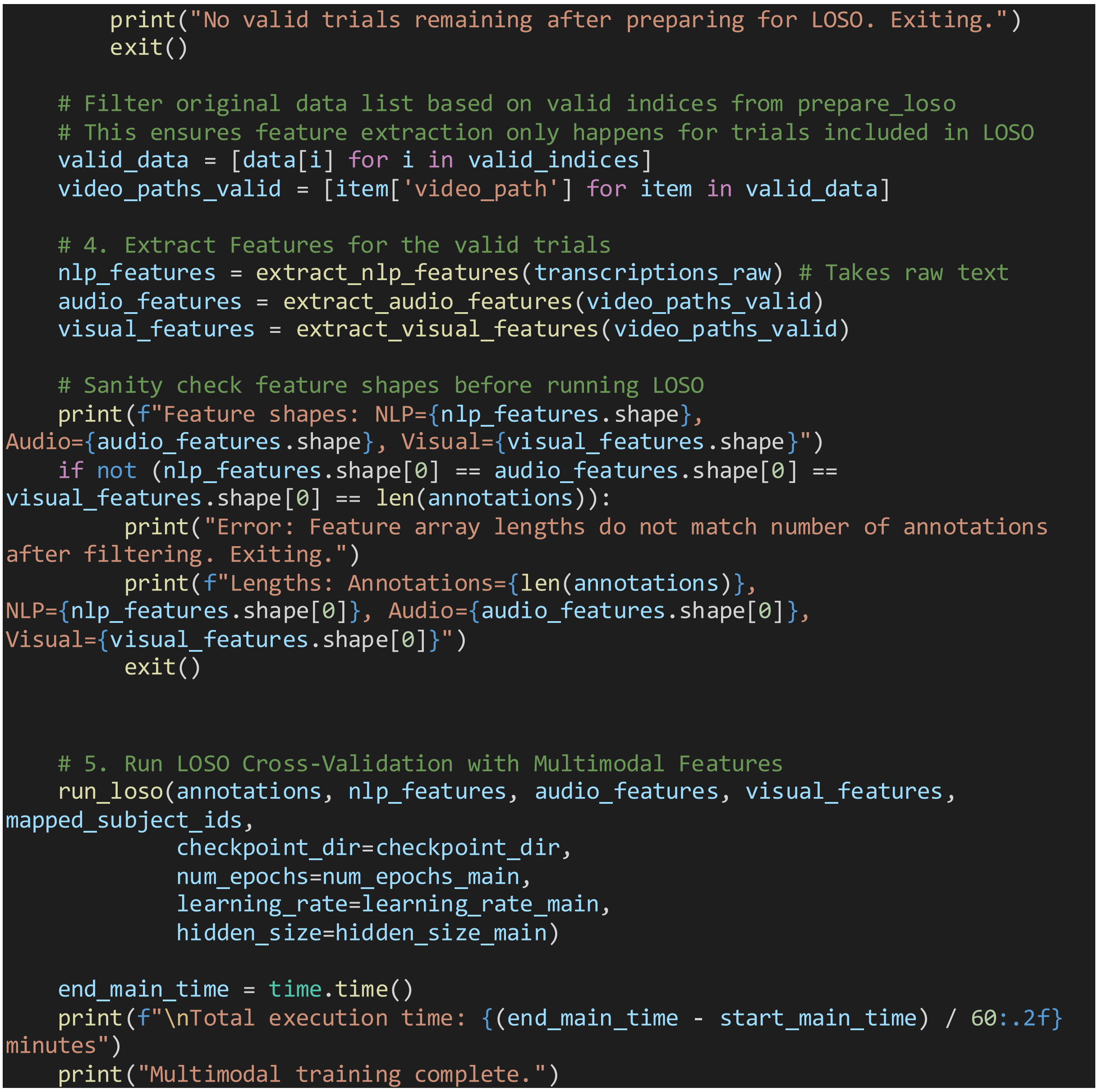
```
    assert len(nlp_features) == n_samples, f"NLP features length mismatch:
{len(nlp_features)} vs {n_samples}"
    assert len(audio_features) == n_samples, f"Audio features length mismatch:
{len(audio_features)} vs {n_samples}"
    assert len(visual_features) == n_samples, f"Visual features length mismatch:
{len(visual_features)} vs {n_samples}"
    assert len(subject_ids) == n_samples, f"Subject IDs length mismatch:
{len(subject_ids)} vs {n_samples}"


    fold_num = 0
    for train_index, test_index in loso.split(nlp_features, annotations,
groups=subject_ids):
        fold_num += 1
        fold_accuracies_seeds = []
        fold_f1s_seeds = []
        fold_losses_seeds = {} # Store losses per seed {seed: [epoch_losses]}

        test_subject = np.unique(np.array(subject_ids)[test_index])[0]
        print(f"\n--- Fold {fold_num}/{loso.get_n_splits(groups=subject_ids)}:
Testing on Subject {test_subject} ---")


        # Split data for this fold
        nlp_train, nlp_test = nlp_features[train_index], nlp_features[test_index]
        audio_train, audio_test = audio_features[train_index],
audio_features[test_index]
        visual_train, visual_test = visual_features[train_index],
visual_features[test_index]
        labels_train, labels_test = annotations[train_index],
annotations[test_index]

        # Check if train or test set is empty for this fold (can happen with LOSO
if subject has few samples)
        if len(labels_train) == 0 or len(labels_test) == 0:
             print(f"Warning: Skipping Fold {fold_num} due to empty train
({len(labels_train)}) or test ({len(labels_test)}) set.")
             continue

        # Get feature dimensions dynamically
        nlp_dim = nlp_train.shape[1]
        audio_dim = audio_train.shape[1]
        visual_dim = visual_train.shape[1]

        for seed in range(num_seeds):
            print(f"  Seed {seed + 1}/{num_seeds}")
            torch.manual_seed(seed)
            np.random.seed(seed) # Also seed numpy if any random operations happen
there

            # Model initialization, optimizer, and loss function
            model = MultimodalDeceptionModel(
                nlp_input_size=nlp_dim,
                audio_input_size=audio_dim,
                visual_input_size=visual_dim,
                hidden_size=hidden_size,
                num_classes=num_classes
            ).to(DEVICE)
```



```
            optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
            criterion = nn.CrossEntropyLoss().to(DEVICE)

            # Define checkpoint file name
            checkpoint_file = os.path.join(
                checkpoint_dir, f"fold_{fold_num}_seed_{seed + 1}.pth")

            # Check if checkpoint exists and load it
            start_epoch = 0
            if os.path.exists(checkpoint_file):
                try:
                    checkpoint = torch.load(checkpoint_file, map_location=DEVICE)
                    model.load_state_dict(checkpoint['model_state_dict'])
                    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
                    start_epoch = checkpoint['epoch']
                    last_loss = checkpoint.get('loss', 'N/A') # Get last saved
loss if available
                    print(f"    Resuming training from checkpoint
{checkpoint_file} at epoch {start_epoch} (Last saved loss: {last_loss})")
                except Exception as e:
                     print(f"    Warning: Could not load checkpoint
{checkpoint_file}: {e}. Starting from scratch.")
                     start_epoch = 0



            # Train and evaluate the model
            epoch_losses = []
            best_f1_seed = -1.0 # Track best F1 for this seed run

            for epoch in range(start_epoch, num_epochs):
                accuracy, f1, loss = train_evaluate(
                    model,
                    nlp_train, audio_train, visual_train, labels_train,
                    nlp_test, audio_test, visual_test, labels_test,
                    optimizer, criterion, DEVICE, epoch)

                epoch_losses.append(loss)

                # Optional: Save checkpoint only if F1 improves for this seed
                if f1 > best_f1_seed:
                     best_f1_seed = f1
                     torch.save({
                        'epoch': epoch + 1,
                        'model_state_dict': model.state_dict(),
                        'optimizer_state_dict': optimizer.state_dict(),
                        'loss': loss,
                        'f1': f1, # Save F1 score in checkpoint
                        'accuracy': accuracy,
                     }, checkpoint_file)
                     # print(f"      Epoch {epoch + 1}/{num_epochs} - Loss:
{loss:.4f}, Acc: {accuracy:.4f}, F1: {f1:.4f} (Checkpoint Saved)")
                #else:
                     # print(f"      Epoch {epoch + 1}/{num_epochs} - Loss:
{loss:.4f}, Acc: {accuracy:.4f}, F1: {f1:.4f}")



            # Store results from the *last* epoch for this seed
            # (Alternatively, load the best checkpoint and evaluate on that)
            # For simplicity, using last epoch results here:
            fold_accuracies_seeds.append(accuracy)
            fold_f1s_seeds.append(f1)
            fold_losses_seeds[seed + 1] = epoch_losses


        # Average metrics across seeds for this fold
        if fold_accuracies_seeds: # Check if any seeds ran successfully
             avg_fold_accuracy = np.mean(fold_accuracies_seeds)
             avg_fold_f1 = np.mean(fold_f1s_seeds)
             all_accuracies.append(avg_fold_accuracy)
             all_f1s.append(avg_fold_f1)
             # Storing all epoch losses per seed per fold can be large - maybe
just avg loss?
             # For now, let's store the list of losses for the last epoch of each
seed
             last_epoch_losses = [losses[-1] for losses in
fold_losses_seeds.values() if losses]
             all_losses.append(np.mean(last_epoch_losses) if last_epoch_losses
else float('nan'))


             print(f"  Fold {fold_num} Average (across seeds) - Accuracy:
{avg_fold_accuracy:.4f}, F1: {avg_fold_f1:.4f}")
        else:
             print(f"  Fold {fold_num} - No successful seed runs.")


    # Overall results across folds
    if all_accuracies:
         overall_avg_accuracy = np.mean(all_accuracies)
         overall_avg_f1 = np.mean(all_f1s)
         print(f"\n--- Overall LOSO Results ---")
         print(f"Overall Average Accuracy: {overall_avg_accuracy:.4f}")
         print(f"Overall Average F1-score: {overall_avg_f1:.4f}")
         # print(f"Average last epoch losses across folds: {all_losses}")
    else:
         print("\n--- No folds completed successfully. Cannot calculate overall
results. ---")

#--- 8. HierarchicalSpatioTemporalAttention ---
class HierarchicalSpatioTemporalAttention(nn.Module):
    """Hierarchical Spatio-Temporal Attention module for video processing."""
    def __init__(self, input_size, hidden_size, num_heads=8, dropout=0.1):
        super(HierarchicalSpatioTemporalAttention, self).__init__()

        # Multi-head attention for spatial features
        self.spatial_attention = nn.MultiheadAttention(
            embed_dim=input_size,
            num_heads=num_heads,
            dropout=dropout
        )

        # Temporal attention layers for different scales
        self.temporal_attention1 = nn.MultiheadAttention(
            embed_dim=input_size,
            num_heads=num_heads,
            dropout=dropout
        )
        self.temporal_attention2 = nn.MultiheadAttention(
            embed_dim=input_size,
            num_heads=num_heads,
            dropout=dropout
        )

        # LSTM for temporal encoding
        self.temporal_encoder = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=2,
            batch_first=True,
            bidirectional=True,
            dropout=dropout
        )

        # Layer normalization
        self.layer_norm1 = nn.LayerNorm(input_size)
        self.layer_norm2 = nn.LayerNorm(input_size)
        self.layer_norm3 = nn.LayerNorm(input_size)

        # Dropout
        self.dropout = nn.Dropout(dropout)

        # Projection layers
        self.projection = nn.Linear(hidden_size * 2, input_size)

    def forward(self, x):
        # x shape: (batch_size, num_frames, num_features)
        batch_size, num_frames, num_features = x.size()

        # Reshape for spatial attention
        spatial_x = x.view(batch_size * num_frames, 1, num_features)
        spatial_x = spatial_x.transpose(0, 1)  # (1, batch_size * num_frames,
num_features)

        # Apply spatial attention
        spatial_out, _ = self.spatial_attention(spatial_x, spatial_x, spatial_x)
        spatial_out = spatial_out.transpose(0, 1)  # (batch_size * num_frames, 1,
num_features)
        spatial_out = spatial_out.view(batch_size, num_frames, num_features)
        spatial_out = self.layer_norm1(x + self.dropout(spatial_out))

        # First temporal scale (original frame rate)
        temporal_out1, _ = self.temporal_attention1(
            spatial_out.transpose(0, 1),
            spatial_out.transpose(0, 1),
            spatial_out.transpose(0, 1)
        )
        temporal_out1 = temporal_out1.transpose(0, 1)
        temporal_out1 = self.layer_norm2(spatial_out +
self.dropout(temporal_out1))

        # Second temporal scale (downsampled)
        if num_frames > 1:
            downsampled = temporal_out1[:, ::2, :]  # Simple downsampling
            temporal_out2, _ = self.temporal_attention2(
                downsampled.transpose(0, 1),
```



```
                downsampled.transpose(0, 1),
                downsampled.transpose(0, 1)
            )
            temporal_out2 = temporal_out2.transpose(0, 1)
            temporal_out2 = self.layer_norm3(downsampled +
self.dropout(temporal_out2))

            # Upsample and combine
            temporal_out2 = F.interpolate(
                temporal_out2.transpose(1, 2),
                size=num_frames,
                mode='linear',
                align_corners=False
            ).transpose(1, 2)
            temporal_out = temporal_out1 + temporal_out2
        else:
            temporal_out = temporal_out1

        # Final temporal encoding with LSTM
        lstm_out, _ = self.temporal_encoder(temporal_out)
        lstm_out = self.projection(lstm_out)

        return lstm_out



# --- 9. Main Execution Block ---
if __name__ == "__main__":
    start_main_time = time.time()

    # --- Configuration ---
    # !!! ADJUST THESE PATHS TO YOUR DATASET LOCATION !!!
    data_dir = 'Real-life_Deception_Detection_2016' # Example path
    annotation_file = "Real-
life_Deception_Detection_2016\Annotation\All_Gestures_Deceptive and Truthful.csv"
    checkpoint_dir = "multimodal_checkpoints"
    num_epochs_main = 50  # Adjust number of epochs
    learning_rate_main = 0.001
    hidden_size_main = 128 # Hidden dimension for feature processing/fusion

    # --- Workflow ---
    # 1. Load Data (Paths, Annotations, Transcriptions)
    data = load_data(data_dir, annotation_file)

    if not data:
        print("No data loaded. Exiting.")
        exit()

    # 2. Identify Subjects (Mandatory Facial Recognition)
    subject_mapping = identify_subjects_facial_recognition(data)
    # print("Subject Mapping (Facial Recognition):", subject_mapping) # Optional
print

    # 3. Prepare Data for LOSO (Get filtered annotations, raw transcriptions,
mapped IDs)
    annotations, transcriptions_raw, mapped_subject_ids, valid_indices =
prepare_loso(data, subject_mapping)

    if len(annotations) == 0:
```



```
        print("No valid trials remaining after preparing for LOSO. Exiting.")
        exit()

    # Filter original data list based on valid indices from prepare_loso
    # This ensures feature extraction only happens for trials included in LOSO
    valid_data = [data[i] for i in valid_indices]
    video_paths_valid = [item['video_path'] for item in valid_data]

    # 4. Extract Features for the valid trials
    nlp_features = extract_nlp_features(transcriptions_raw) # Takes raw text
    audio_features = extract_audio_features(video_paths_valid)
    visual_features = extract_visual_features(video_paths_valid)

    # Sanity check feature shapes before running LOSO
    print(f"Feature shapes: NLP={nlp_features.shape},
Audio={audio_features.shape}, Visual={visual_features.shape}")
    if not (nlp_features.shape[0] == audio_features.shape[0] ==
visual_features.shape[0] == len(annotations)):
         print("Error: Feature array lengths do not match number of annotations
after filtering. Exiting.")
         print(f"Lengths: Annotations={len(annotations)},
NLP={nlp_features.shape[0]}, Audio={audio_features.shape[0]},
Visual={visual_features.shape[0]}")
         exit()



    # 5. Run LOSO Cross-Validation with Multimodal Features
    run_loso(annotations, nlp_features, audio_features, visual_features,
mapped_subject_ids,
             checkpoint_dir=checkpoint_dir,
             num_epochs=num_epochs_main,
             learning_rate=learning_rate_main,
             hidden_size=hidden_size_main)

    end_main_time = time.time()
    print(f"\nTotal execution time: {(end_main_time - start_main_time) / 60:.2f}
minutes")
    print("Multimodal training complete.")
```

OUTPUT:

--- Overall LOSO Results ---

Overall Average Accuracy: 0.6249

Overall Average F1-score: 0.6200