Final Project Assignment #2:

1. Group ID : sp25_DsoDusKan

2. File naming convention:

i) Image dataset - sp25_DsoDusKan_data01_prep.ipynb

ii) Audio dataset - sp25_DsoDusKan_data02_prep.ipynb

ii) Text dataset - sp25_DsoDusKan_data03_prep.ipynb

Image Dataset

Introduction

This report outlines the steps taken to preprocess and convert the **AI-Generated vs. Real Images Dataset** into PyTorch tensors. The dataset consists of images labeled as either AI-generated or real, and the goal is to prepare the data for training a machine learning model. The process involves data loading, cleaning, visualization, imbalance checking, and finally, converting the images and labels into PyTorch tensors.

1. Dataset Overview

The dataset, **AI-Generated vs. Real Images Datasets**, is loaded from Hugging Face's datasets library. It contains **152,710 images** with corresponding labels:

- **Label 0**: AI-Art
- **Label 1**: Real Art

The dataset is stored in a DatasetDict object with a single split (train).

2. Data Loading

The dataset is loaded using the `load_dataset` function from the datasets library. Here's the code snippet:

```
# Import Image dataset from Hugging Face
data01 = load_dataset("Hemg/AI-Generated-vs-Real-Images-Datasets")
print(data01)
```

Output:

```
DatasetDict({
    train: Dataset({
        features: ['image', 'label'],
        num_rows: 152710
    })
})
```

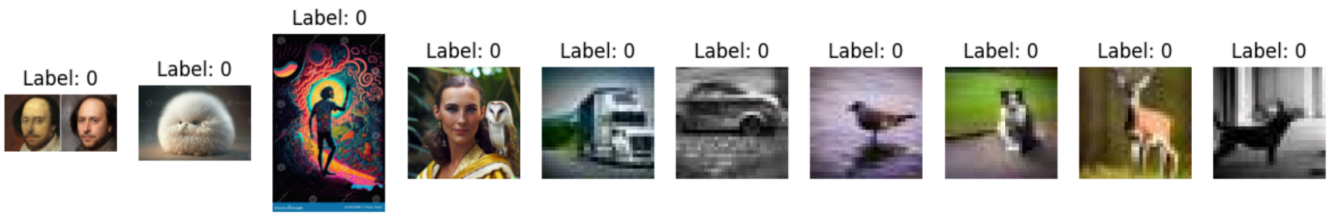
3. Data Visualization

To understand the dataset, we visualize a few samples using matplotlib. The `visualize_images` function displays images along with their labels.

```
# Visualize few sample images
def visualize_images(dataset, num_of_samples):
  figure, axes = plt.subplots(1, num_of_samples, figsize=(15,5))
  for index in range(num_of_samples):
    image = dataset[index]["image"]
    label = dataset[index]["label"]
    axes[index].imshow(image)
    axes[index].set_title(f"Label: {label}")
    axes[index].axis("off")
  plt.show()

visualize_images(data01["train"], 10)
```

Output:

- Displays 10 images with their labels (0 or 1).



4. Data Subsetting

Before continuing with further data processing like Cleaning and Tensorization, we faced an issue with CPU memory usage and tried using Google's T4 GPU, with no luck. Our kernel continued to crash, hence due to memory constraints, we created a subset of **10,000 images** by shuffling the dataset and selecting the first 10,000 samples.

```
# Shuffle the dataset
shuffled_data01 = data01["train"].shuffle(seed=9)

# Create a subset of the dataset by selected rows
sub_data01 = shuffled_data01.select(range(10000))
print(sub_data01)
```

Output:

```
Dataset({
    features: ['image', 'label'],
    num_rows: 10000
})
```

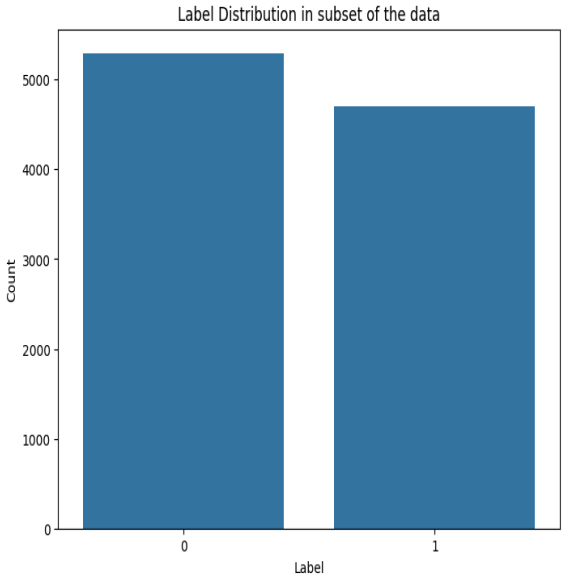
5. Checking for Data Imbalance

We checked the distribution of labels to ensure the dataset is balanced. The `check_imbalance` function calculates the count of each label.

```
# Function to check for imbalance in Target column
def check_imbalance(df, flag=1):
  ''' Inputs to function:
  df   - dataset
  flag - 0 for whole dataset, 1 for subset
  '''
  if flag == 0:
    labels = [i["label"] for i in df["train"]]
  else:
    labels = [i["label"] for i in df]

  label_counts = pd.Series(labels).value_counts()
  print("Target Distribution:")
  print(label_counts)
  return label_counts

# Checking class imbalance in the subset
sub_label_counts = check_imbalance(sub_data01)

# Visualize the label distribution
plt.figure(figsize=(8, 6))
sns.barplot(x=sub_label_counts.index, y=sub_label_counts.values)
plt.xlabel("Label")
plt.ylabel("Count")
plt.title("Label Distribution in subset of the data")
plt.show()
```

Output:

```
Target Distribution:
0    5295
1    4705
Name: count, dtype: int64
```



- ➔ While the imbalance is not extreme, it could still affect the performance of your machine learning model, especially if the model is biased toward the majority class(AI-Art image).
- ➔ To perform oversampling or under-sampling on images, they first need to be converted into numerical format. Then, they can be resampled and converted back to images.

Class Weighting

Instead of resampling, we decided to adjust the class weights in our model to give more importance to the minority class during training. This is the simplest approach and avoids modifying the dataset. Most machine learning frameworks (e.g., PyTorch, TensorFlow) support class weighting.

The dataset has to be checked for potential missing/corrupted images and converted into Tensor format, ahead of performing class weighting.

6. Data Cleaning

An error occurred while Tensorizing the images, because they had different numbers of channels. Some images are RGB (3 channels, e.g., [3, 224, 224]), while others are grayscale (1 channel, e.g., [1, 224, 224]). PyTorch's torch.stack function requires all tensors to have the same shape.

To fix this issue, we had to ensure that all images are converted to the same number of channels (e.g., RGB with 3 channels) before stacking them into a single tensor.

To ensure the dataset is free of corrupted images, we implemented a function `clean_dataset`. This function converts images to RGB format and skips any corrupted files.

```
# Clean the dataset
def clean_dataset(image_dataset):
    clean_data = []
    corrupt_count = 0

    for sample in image_dataset:
        try:
            # Attempt to load the image
            image = sample["image"]
            label = sample["label"]

            # Ensure the image is valid by converting it to RGB (if not already)
            if image.mode != "RGB":
                image = image.convert("RGB")

            # Append the clean data
            clean_data.append({"image": image, "label": label})
        except Exception as e:
            print(f"Corrupted image found: {e}")
            corrupt_count += 1

    print(f"Total corrupted images: {corrupt_count}")
    print(f"Total clean samples: {len(clean_data)}")
    return clean_data

cleaned_data = clean_dataset(sub_data01)
```

Output:

```
Total corrupted images: 0
Total clean samples: 10000
```

7. Converting Data into Tensors

The cleaned dataset is converted into PyTorch tensors for model training. We used the `torchvision.transforms` module to resize images and convert them to tensors.

```
# Check if GPU is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
print(f"Using device: {device}")

# Defining Transformation instance
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize images to a fixed size
    transforms.ToTensor(),          # Convert PIL image to PyTorch tensor
])

# Function to convert image dataset to PyTorch tensors
def dataset_to_tensors(dataset, transform, device, batch_size=32,
save_dir="batches"):
    os.makedirs(save_dir, exist_ok=True)  # Create directory to save batches

    images_batch = []
    labels_batch = []

    for i, sample in enumerate(dataset):
        image = sample["image"]
        label = sample["label"]

        # Apply transformations
        image = transform(image)

        images_batch.append(image)
        labels_batch.append(label)

        # Save batch when batch_size is reached or at the end of the dataset
        if (i + 1) % batch_size == 0 or (i + 1) == len(dataset):
            # Convert batch to tensors
            images_tensor = torch.stack(images_batch).to(device)
            labels_tensor = torch.tensor(labels_batch).to(device)

            # Save batch to disk
            batch_filename = os.path.join(save_dir, f"batch_{i //
batch_size}.pt")
            torch.save({"images": images_tensor, "labels": labels_tensor},
batch_filename)
            print(f"Saved {batch_filename}")

            # Reset batch lists
            images_batch, labels_batch = [], []

# Process the dataset in batches and save incrementally
dataset_to_tensors(cleaned_data, transform, device, batch_size=100,
save_dir="batches")
```

Few key points are considered/learned while writing this code:

GPU Availability: The script checks GPU availability using PyTorch and assigns computations to either a CUDA-enabled GPU or the CPU for optimal performance.

Image Transformations: A transformation pipeline is defined using transforms.compose() from PyTorch torchvision to resize images to 224x224 pixels and convert them to PyTorch tensors.

Batch Processing and Saving: The `dataset_to_tensors()` function processes the dataset incrementally, applying transformations, accumulating images and labels into batches, and saving each batch to disk as a .pt file when the specified batch size is reached or the dataset ends. We updated the `batch_size=100` instead of the default size of 32 to keep the number of batches to an optimum number. The images dataset is splitted into 99 sub-sets.

Memory Efficiency: The script saves batches incrementally and resets memory buffers after each save to avoid excessive memory usage, making it suitable for large datasets.

Modularity and Reusability: The script is configurable with parameters for batch size and output directory, supporting easy adaptation for different datasets or processing environments.

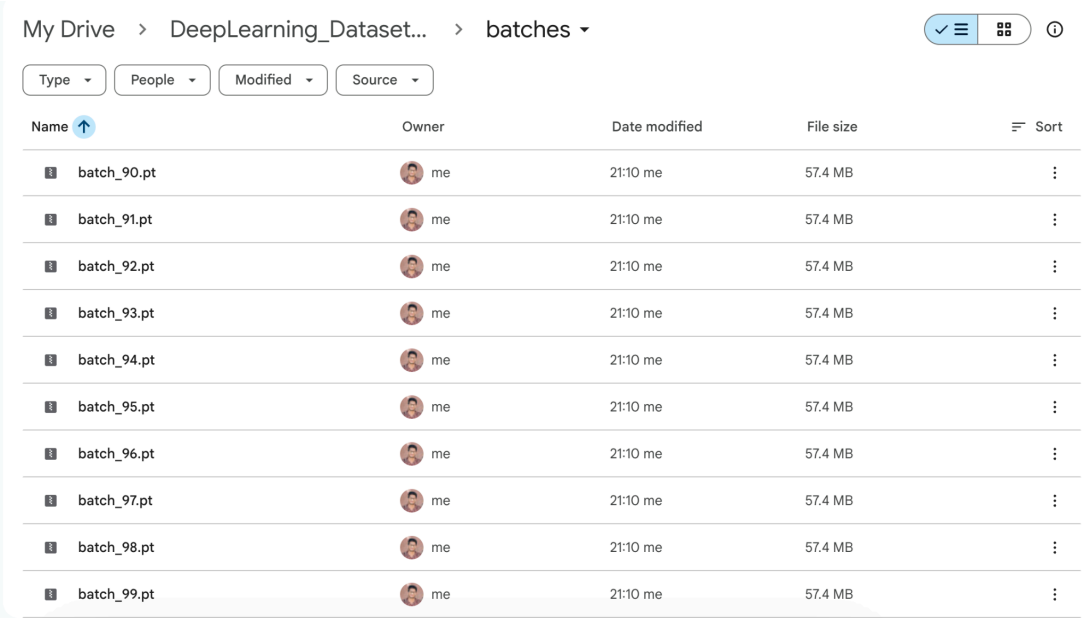
8. Saving the Tensor datafiles for future use

The following script transfers batch files from local runtime to Google Drive. Batch File Transfer Process iterates through batch files, moves them to Google Drive, and provides a confirmation message.

```
# Mount Google Drive
drive.mount('/content/drive')

# Define the destination directory in Google Drive
drive_path = '/content/drive/MyDrive/DeepLearning_Datasetes/batches/'

# Create the directory if it doesn't exist
os.makedirs(drive_path, exist_ok=True)

# Move all batch files to Google Drive
for batch_file in os.listdir("batches"):
   shutil.move(os.path.join("batches", batch_file), drive_path + batch_file)

print(f"All batch files saved to Google Drive at: {drive_path}")
```

Output:

```
All batch files saved to Google Drive at:
/content/drive/MyDrive/DeepLearning_Datasetes/batches/
```



Link to Google Colab file - sp25_DsoDusKan_data01_prep.ipynb

Our Learnings

- While dealing with images, when converting into tensor format, we need to ensure all the images are either in RGB format or Grayscale format.
- Batch processing came to the rescue when we are struggling to figure out why the kernel keeps crashing. It really made sense, after trying it out.
- Having a CUDA enabled device makes things easier when dealing with Tensors. Apple Macbook had to stress everything on the unified memory.
- Need to check on overriding the default Timeout setting to keep the kernel from crashing.

Text Dataset

Introduction & Dataset Overview

This report details the process of preparing a dataset for deep learning experimentation in Google Colab. The dataset used in this round is **"Sentiment Analysis for Mental Health"** from Kaggle, which consists of textual statements classified into different mental health categories: **Normal, Depression, Suicidal, Anxiety, Bipolar, Stress, and Personality Disorder**.

The **ML problem** defined for this dataset is **text classification**, where the goal is to train a deep learning model to predict the mental health status based on a given textual statement.

Data Loading & Cleaning

The dataset was downloaded directly from Kaggle using `kagglehub`:

```
import kagglehub
path =
kagglehub.dataset_download("suchintikasarkar/sentiment-analysis-for-mental-health
")
df = pd.read_csv(path + "/Combined Data.csv")
```

Cleaning & Preprocessing

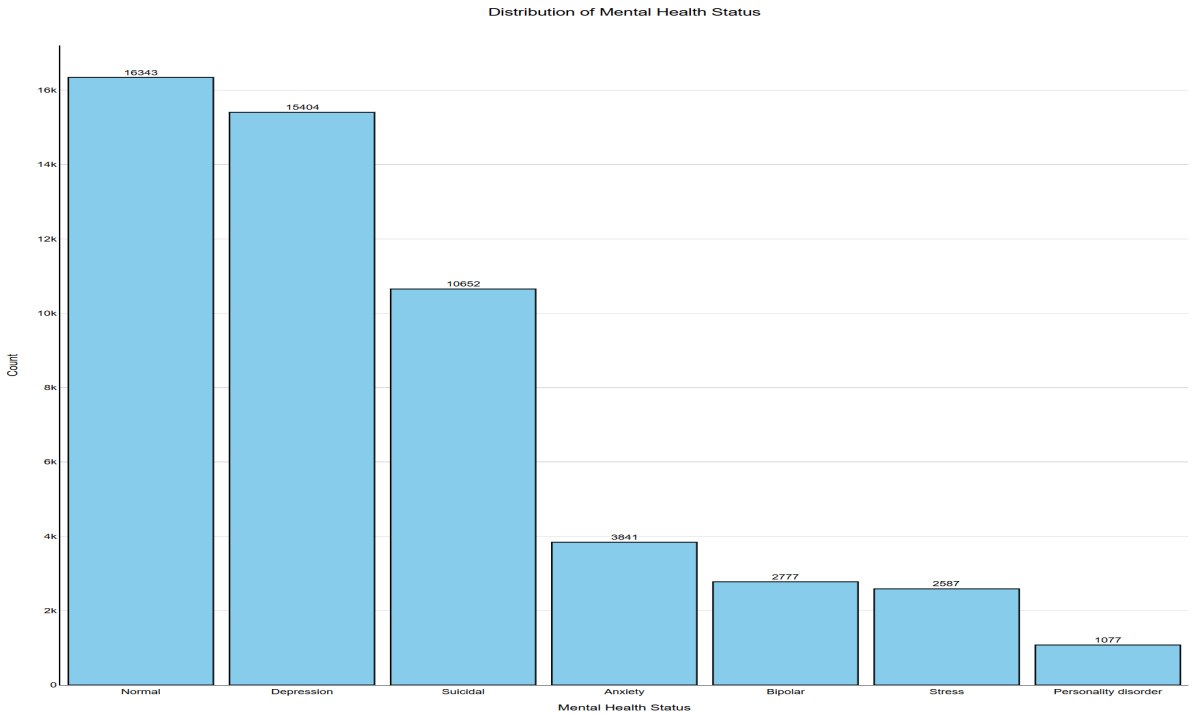
Steps taken to clean the dataset:

- **Removed unnecessary columns** (e.g., `Unnamed: 0`)
- **Checked for missing values** in the `statement` and `status` columns
- **Dropped rows where `statement` was missing**

```
df = df.drop(columns=['Unnamed: 0'])
df = df.dropna(subset=["statement"]).reset_index(drop=True)
```

Balancing the Dataset

Before Balancing

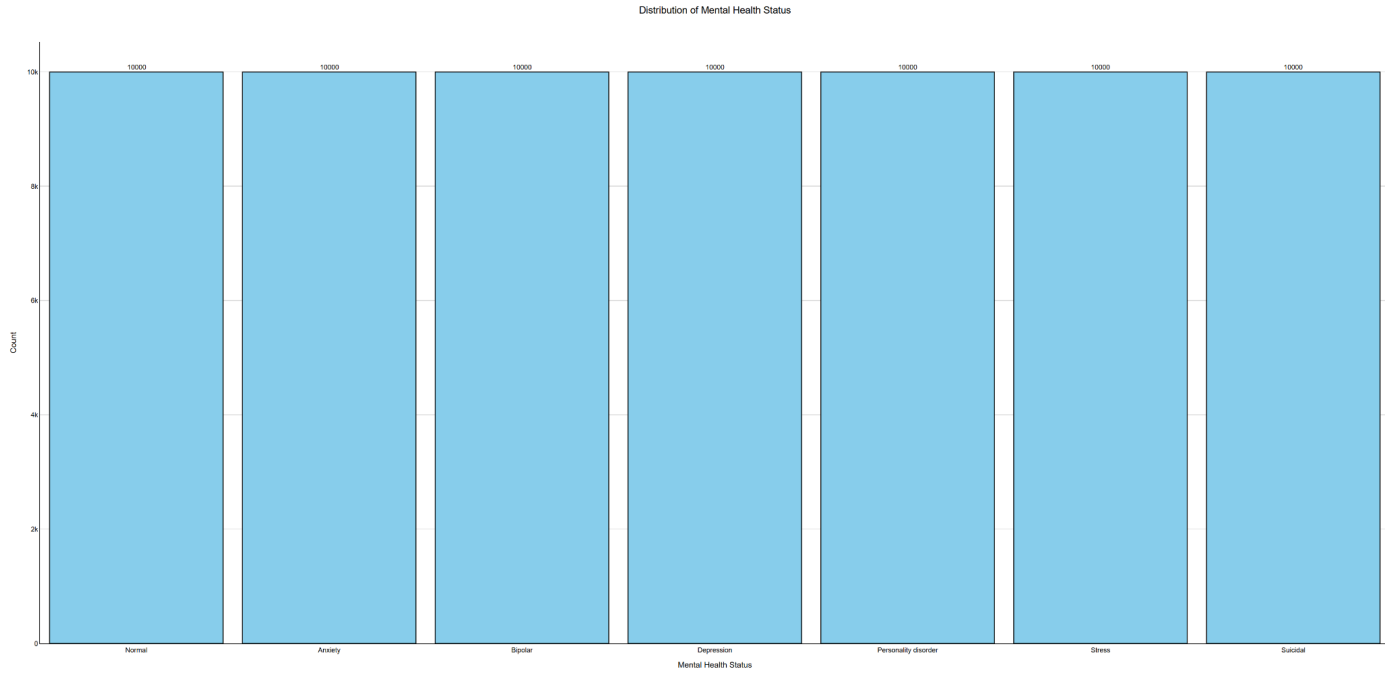


Since the dataset was highly imbalanced, a **hybrid approach (undersampling & oversampling)** was applied:

- **Undersampled the "Normal" class** to **10,000 samples**
- **Oversampled all minority classes** to **10,000 samples each** using `replace=True`

```
# Undersample the majority class ("Normal")
df_majority = df[df["status"] == "Normal"].sample(10000, random_state=30)

# Extract minority classes
df_minority = df[df["status"] != "Normal"]

# Oversample each minority class to match 10,000 samples
df_minority_balanced = df_minority.groupby("status").apply(lambda x:
x.sample(10000, replace=True, random_state=33)).reset_index(drop=True)

# Combine undersampled majority and oversampled minority data
df_balanced = pd.concat([df_majority, df_minority_balanced])

# Checking new class distribution
print(df_balanced["status"].value_counts())
```

After balancing



All **classes have 10,000 samples**, ensuring a fair model training process.

This combination was chosen because only undersampling would result in significant data loss, and only oversampling could lead to overfitting due to excessive duplication of limited minority class samples. By combining both methods, we preserved diversity in the dataset while ensuring a balanced representation across all classes.

Convert Dataset into Tensor Format

Since the dataset consists of **textual data**, tokenization and tensor conversion were necessary.

Why LabelEncoder Before Tensorization?

`LabelEncoder` was used to convert categorical text labels into numerical values because deep learning models require numerical inputs. This step ensures that the labels are in a machine-readable format before tensor conversion, avoiding complications in model training.

```
label_encoder = LabelEncoder()
df_balanced['status'] = label_encoder.fit_transform(df_balanced['status'])

df_balanced.head()
```

Why `bert-base-uncased`?

The **BERT-based uncased model** was chosen because:

- It has been pre-trained on a large dataset and captures deep contextual meanings of text.
- The uncased version is used to avoid distinctions between uppercase and lowercase words, which is useful for informal text data.

Why `max_length=128`?

A maximum length of 128 was selected based on:

- Keeping memory efficiency in mind while ensuring that most sentences retain their full context.
- Avoiding unnecessary truncation while keeping computations manageable.

Why `torch.long` for Labels?

Labels were converted using `torch.long` because:

- PyTorch classification models require integer labels in **long tensor format**.
- Ensures compatibility with loss functions like `CrossEntropyLoss`, which expects long-type labels.

Why Use `Dataset` & `DataLoader`?

The combination of `Dataset` and `DataLoader` was chosen because:

- `Dataset` allows easy **custom data processing**, tokenization, and label retrieval.
- `DataLoader` enables **batch processing, shuffling, and parallel data loading**, which is necessary for efficient deep learning training.

A **custom dataset class** was created to **tokenize** the text and convert it into tensors:

```
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

class SentimentDataset(Dataset):
    def __init__(self, texts,labels,tokenizer,max_length = 128):
      self.texts = texts
      self.labels = labels
      self.tokenizer = tokenizer
      self.max_length = max_length

    def __len__(self):
      return len(self.texts)
```

```
    def __getitem__(self, idx):
      text = str(self.texts[idx])
      label = torch.tensor(self.labels[idx],dtype=torch.long)


      encoding = self.tokenizer.encode_plus(
          text,
          padding='max_length',
          truncation=True,
          max_length=self.max_length,
          return_tensors='pt'
      )

      return {
          'input_ids': encoding['input_ids'].squeeze(),
          'attention_mask': encoding['attention_mask'].squeeze(),
          'labels': label
      }

texts = df_balanced["statement"].tolist()
labels = df_balanced["status"].tolist()


dataset = SentimentDataset(texts, labels, tokenizer)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

batch = next(iter(dataloader))
print(batch)
```

5. Save Processed Data for Future Use

To ensure that the processed tensors can be reused, they were saved to **Google Drive**:

```
torch.save(dataset, "/content/drive/MyDrive/ASDS5306/sentiment_dataset.pt")
torch.save(dataloader,
"/content/drive/MyDrive/ASDS5306/sentiment_dataloader.pt")

joblib.dump(label_encoder, "/content/drive/MyDrive/ASDS5306/label_encoder.pkl")

print("Tensorized data saved in Google Drive!")
```

This allows **future notebooks to directly load and use the tensorized data**

Links to Notebook & Data Files:

- **Notebook:** Google Colab Link
- **Processed Dataset:** Download sentiment_dataset.pt
- **DataLoader:** Download sentiment_dataloader.pt
- **Label Encoder:** Download label_encoder.pkl

Audio Dataset

1. Introduction & Dataset Overview

This project focuses on preparing audio datasets for deep learning experimentation to recognize emotions from speech. The datasets used are:

- **RAVDESS**: A high-quality emotional speech dataset containing neutral, calm, happy, sad, angry, fearful, disgusted, and surprised emotions.
- **CREMA-D**: Contains emotional audio clips from professional actors, covering six emotions.
- **TESS**: Composed of recordings by two female speakers, covering seven emotions.
- **SAVEE**: Features male speakers expressing six emotions.

Preprocessing

Emotion Mapping and File Extraction

```
ravdess_directory_list = os.listdir(Ravdess)

file_emotion = []
file_path = []
for dir in ravdess_directory_list:
   # as there are 20 different actors in our previous directory we need to extract
files for each actor.
   actor = os.listdir(Ravdess + dir)
   for file in actor:
       # Check if the file name follows the expected pattern
       if len(file.split('-')) >= 3:
           part = file.split('.')[0]
           part = part.split('-')
            # third part in each file represents the emotion associated to that file.
           file_emotion.append(int(part[2]))
```

```
            file_path.append(Ravdess + dir + '/' + file)
        else:
            # Handle files with different naming conventions (e.g., skip or print a
warning)
            print(f"Skipping file with unexpected name: {file}")
            # You could also choose to skip the file and continue with the next one:
            # continue

# dataframe for emotion of files
emotion_df = pd.DataFrame(file_emotion, columns=['Emotions'])

# dataframe for path of files.
path_df = pd.DataFrame(file_path, columns=['Path'])
Ravdess_df = pd.concat([emotion_df, path_df], axis=1)

# changing integers to actual emotions.
Ravdess_df.Emotions.replace({1:'neutral', 2:'calm', 3:'happy', 4:'sad', 5:'angry',
6:'fear', 7:'disgust', 8:'surprise'}, inplace=True)
Ravdess_df.head()
```

Emotion labels were derived from audio recordings by iterating through them and using file naming standards. For instance, the emotion is specified by the third part of filenames in RAVDESS.

```
File: 03-01-06-01-02-02-12.wav
Emotion: Fear (mapped from the third segment)
```

Consolidating Data

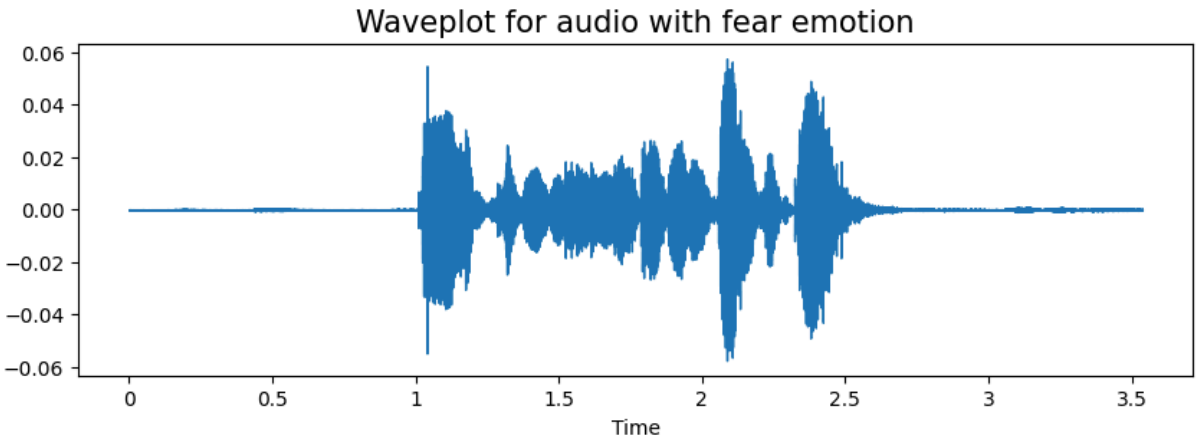
The emotion labels and file paths from all datasets were merged into a single DataFrame, saved as `data_path.csv`. This file serves as a unified dataset for further processing.

```
# creating Dataframe using all the 4 dataframes we created so far.
data_path = pd.concat([Ravdess_df, Crema_df, Tess_df, Savee_df], axis = 0)
data_path.to_csv("data_path.csv",index=False)
data_path.head()
```

	Emotions	Path
0	happy	/root/.cache/kagglehub/datasets/uwrfkaggler/ra...
1	calm	/root/.cache/kagglehub/datasets/uwrfkaggler/ra...
2	angry	/root/.cache/kagglehub/datasets/uwrfkaggler/ra...
3	fear	/root/.cache/kagglehub/datasets/uwrfkaggler/ra...
4	sad	/root/.cache/kagglehub/datasets/uwrfkaggler/ra...

Sample code to create a wave-plot for the emotion '**fear**'

```
emotion='fear'
path = np.array(data_path.Path[data_path.Emotions==emotion])[1]
data, sampling_rate = librosa.load(path)
create_waveplot(data, sampling_rate, emotion)
Audio(path)
```



Data Augmentation

To improve model generalization, synthetic data was generated using the following techniques:

Noise Injection: Random noise added to the signal.

Time Stretching: Slowing down or speeding up the audio.

Pitch Shifting: Changing the pitch without affecting speed.

Shifting: Shifting audio data along the time axis.

Each augmentation method retains the original emotion label to maintain dataset integrity.

Functions to create different augmentations:

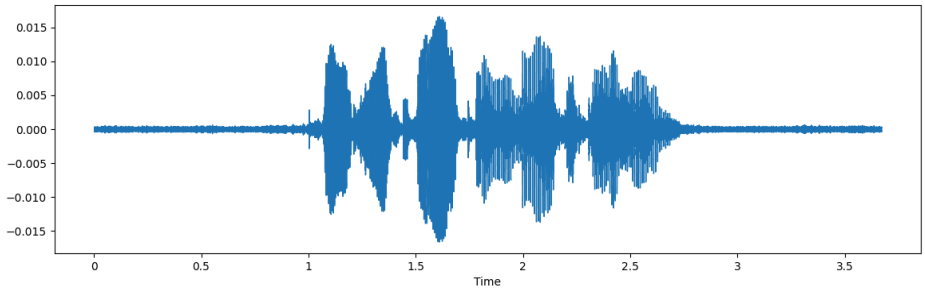
```
def noise(data):
   noise_amp = 0.035*np.random.uniform()*np.amax(data)
```

```
    data = data + noise_amp*np.random.normal(size=data.shape[0])
    return data

def stretch(data, rate=0.8):
    return librosa.effects.time_stretch(data, rate=rate) # Changed: Pass rate as
keyword argument

def shift(data):
    shift_range = int(np.random.uniform(low=-5, high = 5)*1000)
    return np.roll(data, shift_range)

def pitch(data, sampling_rate, n_steps=0.7):
    return librosa.effects.pitch_shift(data, sr=sampling_rate, n_steps=n_steps)
# taking any example and checking for techniques.
path = np.array(data_path.Path)[1]
data, sample_rate = librosa.load(path)
```

Sample code to induce noise into an audio file:

```
x = noise(data)
plt.figure(figsize=(14,4))
librosa.display.waveshow(y=x, sr=sample_rate)
Audio(x, rate=sample_rate)
```

Output: Sound wave after inducing noise.



Feature Extraction

Techniques Used:

1. **MFCC (Mel Frequency Cepstral Coefficients):** Captures essential audio features for speech recognition.
2. **Zero Crossing Rate (ZCR):** Measures the rate at which the signal changes sign.
3. **Chroma Features:** Represents the energy distribution across the 12 pitch classes.
4. **MelSpectrogram:** Provides a time-frequency representation of audio signals.
5. **Root Mean Square (RMS):** Measures signal energy.

Process:

- Each audio file is loaded, and features are extracted using `librosa`.
- Extracted features are stacked into a single array for consistency.

```
def extract_features(data):
    # ZCR
    result = np.array([])
    zcr = np.mean(librosa.feature.zero_crossing_rate(y=data).T, axis=0)
    result=np.hstack((result, zcr)) # stacking horizontally

    # Chroma_stft
    stft = np.abs(librosa.stft(data))
    chroma_stft = np.mean(librosa.feature.chroma_stft(S=stft, sr=sample_rate).T,
axis=0)
    result = np.hstack((result, chroma_stft)) # stacking horizontally

    # MFCC
    mfcc = np.mean(librosa.feature.mfcc(y=data, sr=sample_rate).T, axis=0)
    result = np.hstack((result, mfcc)) # stacking horizontally

    # Root Mean Square Value
    rms = np.mean(librosa.feature.rms(y=data).T, axis=0)
    result = np.hstack((result, rms)) # stacking horizontally

    # MelSpectogram
    mel = np.mean(librosa.feature.melspectrogram(y=data, sr=sample_rate).T,
axis=0)
    result = np.hstack((result, mel)) # stacking horizontally

    return result

def get_features(path):
    # duration and offset are used to take care of the no audio in start and the
ending of each audio files as seen above.
    data, sample_rate = librosa.load(path, duration=2.5, offset=0.6)

    # without augmentation
    res1 = extract_features(data)
    result = np.array(res1)

    # data with noise
    noise_data = noise(data)
    res2 = extract_features(noise_data)
    result = np.vstack((result, res2)) # stacking vertically
```

```
    # data with stretching and pitching
    new_data = stretch(data)
    data_stretch_pitch = pitch(new_data, sample_rate)
    res3 = extract_features(data_stretch_pitch)
    result = np.vstack((result, res3)) # stacking vertically

    return result
```

Tensor Conversion

Reshaping Audio Data

- Audio features were padded or truncated to a consistent shape using the following logic:
- Labels were encoded using `LabelEncoder` to transform emotions into integers.

```
# Load the dataset
data_path = pd.read_csv("data_path.csv")

# Function to extract MFCC features from audio files
def extract_mfcc(file_path, n_mfcc=13, max_pad_len=174):
   try:
       # Load audio file
       audio, sr = librosa.load(file_path, sr=None)
       # Extract MFCC features
       mfccs = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=n_mfcc)
       # Pad or truncate to ensure consistent shape
       if mfccs.shape[1] < max_pad_len:
           pad_width = max_pad_len - mfccs.shape[1]
           mfccs = np.pad(mfccs, pad_width=((0, 0), (0, pad_width)), mode='constant')
       else:
           mfccs = mfccs[:, :max_pad_len]
       return mfccs
   except Exception as e:
       #print(f"Error processing {file_path}: {e}")
       return None
```

```
# Extract features and labels
features = []
labels = []

for index, row in data_path.iterrows():
    file_path = row['Path']
    emotion = row['Emotions']
    mfccs = extract_mfcc(file_path)
    if mfccs is not None:
        features.append(mfccs)
        labels.append(emotion)

# Convert labels to numerical format
label_encoder = LabelEncoder()
labels_encoded = label_encoder.fit_transform(labels)
```

The `torch.utils.data.Dataset` and `DataLoader` classes were used to:

- Batch the dataset.
- Shuffle data for training.

```
# Convert features and labels to PyTorch tensors
features_tensor = torch.tensor(features, dtype=torch.float32)
labels_tensor = torch.tensor(labels_encoded, dtype=torch.long)

# Print shapes to verify
print(f"Features tensor shape: {features_tensor.shape}")
print(f"Labels tensor shape: {labels_tensor.shape}")

# Create a custom PyTorch Dataset
class AudioDataset(Dataset):
    def __init__(self, features, labels):
        self.features = features
        self.labels = labels

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

    def __getitem__(self, idx):
        return self.features[idx], self.labels[idx]

# Create dataset and dataloader
dataset = AudioDataset(features_tensor, labels_tensor)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Below are the learnings that we take away from the process of converting an audio file into pyTorch tensor.

- Converting raw waveforms to tensors using features such as Mel Spectrograms, MFCCs (Mel-Frequency Cepstral Coefficients), or STFT (Short-Time Fourier Transform) is essential for effective model learning.
- Organizing audio tensors into batches while maintaining consistent dimensions is necessary for efficient model training.
- Ensuring that tensors are moved to the correct device (e.g., cuda for GPU or cpu) for faster computation.

Link to Google Colab file - sp25_DsoDusKan_data02_prep.ipynb