# Assignment: Speech Recognition & Dialect Adaptation - Train an ASR Model

## 1. Introduction

Automatic Speech Recognition (ASR) systems have become ubiquitous in modern technology, powering voice assistants, transcription services, and more. However, a significant challenge lies in the variability of human speech, particularly across different accents and dialects. This assignment focuses on building an ASR system that is robust to these variations through dialect adaptation techniques. Additionally, we will explore a simple NLP-based post-processing step to correct common misinterpretations.

## 2. Learning Objectives

Upon completion of this assignment, students will be able to:

- Understand the fundamental concepts of speech recognition and acoustic modeling.
- Identify the challenges posed by dialectal variations in ASR.
- Implement data augmentation techniques for dialect adaptation.
- Train and evaluate an ASR model using a deep learning framework.
- Apply transfer learning to adapt a pre-trained ASR model to a specific dialect.
- Develop a basic NLP model for correcting common ASR errors.
- Evaluate the performance of the ASR system and the NLP correction model.

## 3. Dataset

- **Primary Dataset:**
  Choose a multi-dialectal speech dataset. Consider these options:
  - **Common Voice:** Mozilla's Common Voice project offers speech data in various languages and accents. It's a good option for general-purpose adaptation. (https://commonvoice.mozilla.org/en/datasets)
  - **LibriSpeech:** While primarily American English, it's a well-established dataset for ASR development (http://www.openslr.org/12). You could attempt to adapt it to a specific accent, perhaps by augmenting with synthetic data (see below).
  - **Custom Dataset:** If possible, collecting a small dataset of speech from a specific dialect would be very beneficial. This could involve recording volunteers or using publicly available audio (ensure proper licensing and ethical considerations).

- **Error Correction Dataset:** Use a common text corpus (e.g., Wikipedia, news articles) for training the NLP error correction model. This will give the model a general understanding of word frequencies and context.

## 4. Project Steps

### 4.1. Data Preprocessing and Exploration

- **Data Acquisition:** Download and organize the chosen speech dataset(s).
- **Data Exploration:**

  - Analyze the distribution of accents/dialects in the dataset.
  - Examine the audio characteristics of different dialects (e.g., pitch, speaking rate, pronunciation variations).
  - Identify common phonetic differences between dialects.

- **Data Cleaning:**

  - Remove noise and silence.
  - Normalize volume levels.
  - Handle missing or corrupted data.

- **Feature Extraction:**

- Extract relevant acoustic features from the audio data. Common features include:
  - Mel-Frequency Cepstral Coefficients (MFCCs)
  - Filter Bank Energies (FBANKs)
  - Spectrograms

- Use libraries like librosa in Python for feature extraction:

```
import librosa
import librosa.display
import numpy as np

def extract_mfccs(audio_path, n_mfcc=13):
    """Extracts MFCC features from an audio file."""
    try:
        y, sr = librosa.load(audio_path)
        mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=n_mfcc)
        return mfccs.T  # Transpose for time-major format
    except Exception as e:
        print(f"Error processing {audio_path}: {e}")
        return None

# Example Usage
audio_file = "path/to/your/audio.wav"
mfccs = extract_mfccs(audio_file)

if mfccs is not None:
    print("MFCCs shape:", mfccs.shape)  # (num_frames, n_mfcc)

    # Optional: Visualize MFCCs
    import matplotlib.pyplot as plt
```

```
librosa.display.specshow(mfccs.T, sr=sr, x_axis='time')
plt.colorbar()
plt.title('MFCC')
plt.tight_layout()
plt.show()
```

- **Text Preprocessing:**
  - Clean the text transcriptions (remove punctuation, convert to lowercase).
  - Create a vocabulary of words and characters.
  - Consider using subword tokenization (e.g., SentencePiece, Byte-Pair Encoding) to handle out-of-vocabulary words and improve model generalization.

### 4.2. Data Augmentation (Dialect Adaptation)

To improve the ASR system's performance on different dialects, apply data augmentation techniques:

- **Speed Perturbation:**
  Slightly alter the speed of the audio.
  - Increase or decrease the speed by a small factor (e.g., 0.9, 1.1).
  - Use librosa.effects.time_stretch for this.

- **Pitch Shifting:**
  Shift the pitch of the audio.
  - Increase or decrease the pitch by a few semitones.
  - Use librosa.effects.pitch_shift.

- **Volume Adjustment:** Randomly increase or decrease the volume.
- **Adding Noise:** Add background noise or simulated accent-specific noise. You could create noise profiles from audio samples of different

dialects.
- **SpecAugment:** Apply time and frequency masking to the spectrograms. This can improve robustness.
- **Text-to-Speech (TTS) Synthesis:** If you have a TTS system capable of generating speech in different dialects, use it to synthesize additional training data. This is a more advanced technique but can be very effective.

```
import librosa
import soundfile as sf
import numpy as np

def augment_audio(audio_path, output_path, speed_factor=1.0, pitch_shift=0, volume_factor=1.0, noise_level=0.0):
    """Augments audio with speed change, pitch shift, volume adjustment, and noise addition."""
    try:
        y, sr = librosa.load(audio_path)

        # Speed Perturbation
        if speed_factor != 1.0:
            y = librosa.effects.time_stretch(y, rate=speed_factor)

        # Pitch Shifting
        if pitch_shift != 0:
            y = librosa.effects.pitch_shift(y, sr=sr, n_steps=pitch_shift)

        # Volume Adjustment
        y = y * volume_factor

        # Add Noise
        if noise_level > 0.0:
            noise = np.random.randn(len(y)) * noise_level
            y = y + noise

        # Normalize to prevent clipping
        y = np.clip(y, -1.0, 1.0)
        sf.write(output_path, y, sr)  # Save the augmented audio
        return True
    except Exception as e:
        print(f"Error augmenting {audio_path}: {e}")
        return False


# Example usage
original_audio = "path/to/your/audio.wav"
augmented_audio = "path/to/your/augmented_audio.wav"

augment_audio(original_audio, augmented_audio, speed_factor=0.95, pitch_shift=2, volume_factor=1.2, noise_level=0.01)
```

## 4.3. ASR Model Selection and Training

- **Model Selection:**
  Choose a suitable ASR model architecture. Consider these options:
  - **DeepSpeech2:** A popular end-to-end ASR model based on recurrent neural networks (RNNs) or LSTMs.
  - **Connectionist Temporal Classification (CTC) models:** CTC is a loss function used for training sequence-to-sequence models without explicit alignment. Many ASR architectures are based on CTC.
  - **Transformers:** Transformer-based models (e.g., Whisper) have achieved state-of-the-art results in ASR. The transformers library makes it relatively easy to use these models.
  - **QuartzNet/Jasper:** Convolutational models that are efficient and perform well.

- **Implementation:** Implement the chosen model using a deep learning framework like TensorFlow or PyTorch.
- **Training:**

  - Train the model on the preprocessed and augmented dataset.
  - Use appropriate optimization algorithms (e.g., Adam).
  - Monitor the training progress using appropriate metrics (e.g., Word Error Rate (WER), Character Error Rate (CER)).
  - Implement early stopping to prevent overfitting.

- **Transfer Learning:**
    - If you are using a pre-trained ASR model, fine-tune it on your dialect-specific data. This can significantly improve performance. A very effective approach.
    - The transformers library is ideal for transfer learning with pre-trained models.

```
from transformers import WhisperProcessor, WhisperForConditionalGeneration
from datasets import load_dataset
import torch

# 1. Load pre-trained Whisper model and processor
model_name = "openai/whisper-small"  # Or "whisper-base", "whisper-medium", "whisper-large"
processor = WhisperProcessor.from_pretrained(model_name)
model = WhisperForConditionalGeneration.from_pretrained(model_name)
model.config.forced_decoder_ids = None
model.config.suppress_tokens = []

# Move model to GPU if available
device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)

# 2. Load your audio dataset (replace with your data loading)
# Example using Hugging Face Datasets:
dataset = load_dataset("librispeech_asr", "clean", split="validation[:10]") # Example: first 10 samples of LibriSpeech


def prepare_dataset(batch):
    audio = batch["audio"]

    # Load audio and resample if necessary
    audio_input = librosa.load(audio["path"], sr=audio["sampling_rate"])[0]
    # Convert to input features
    input_features = processor(audio_input, sampling_rate=audio["sampling_rate"], return_tensors="pt").input_features

    # Transcriptions
    batch["input_features"] = input_features
    batch["labels"] = processor.tokenizer(batch["text"]).input_ids
    return batch

dataset = dataset.map(prepare_dataset, remove_columns=list(dataset.features.keys()), batched=False)
dataset = dataset.filter(lambda x: len(x["input_features"]) < 1500)


# 3. Training (Simplified - requires a proper training loop)

from transformers import Seq2SeqTrainer, Seq2SeqTrainingArguments

training_args = Seq2SeqTrainingArguments(
    output_dir="./whisper-finetuned",
    per_device_train_batch_size=16,
    gradient_accumulation_steps=1, # Increase for larger batch sizes
    learning_rate=1e-5,
    warmup_steps=500,
    max_steps=4000,
    fp16=True, # Use mixed precision training for faster training
    evaluation_strategy="steps",
    per_device_eval_batch_size=8,
    predict_with_generate=True,
    log_level="error",
    logging_steps=25,
    save_steps=500,
    eval_steps=500,
    save_total_limit=2,
)
```

```
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=dataset,
    eval_dataset=dataset, # Or use a separate evaluation dataset
    data_collator=lambda data: {
        'input_features': torch.stack([f['input_features'][0] for f in data]),
        'labels': torch.tensor([f['labels'] for f in data])
    },
    tokenizer=processor.feature_extractor,
)


trainer.train()

# 4. Save the fine-tuned model
model.save_pretrained("path/to/your/fine_tuned_model")
processor.save_pretrained("path/to/your/fine_tuned_model")  #Also save the processor
```

**Important Notes on the Code Above:**

- **Dataset Loading:** The load_dataset example is just a starting point. You'll need to adapt it to your specific dataset format. You will likely need to create a custom data loading pipeline.
- **Data Collator:** This is a *very* important part of the training process. The data collator takes a batch of examples and prepares them for the model. The provided example is very basic and may need to be adjusted based on your data.
- **Training Loop:** The code skips over the details of the training loop. A complete training loop will involve iterating over the dataset in batches, calculating the loss, updating the model's parameters, and evaluating the model's performance.
- **GPU:** Training deep learning models without a GPU is very slow. Make sure you have access to a GPU if you want to train your model in a reasonable amount of time. Also, make sure you have set up CUDA correctly.
- **fp16:** Using fp16 (mixed precision training) can significantly speed up training on GPUs that support it.

## 4.4. NLP Error Correction

Develop a small NLP model to correct common misinterpretations made by the ASR system. This acts as a post-processing step.

- **Error Analysis:** Analyze the common types of errors made by your ASR model. Look for patterns, e.g., phonetic confusions ("to" vs. "too" vs. "two"), homophones, and context-dependent errors.

- **Model Selection:** Choose a simple NLP model for error correction. Consider these options:

  - **N-gram Language Model:** Calculate the probabilities of word sequences and correct errors by selecting the most probable sequence.
  - **Conditional Probability-based Correction:** Calculate the probability of a correct word given the ASR output and the surrounding context. This is more sophisticated.
  - **Simple Sequence-to-Sequence Model:** Use a small Transformer or LSTM-based sequence-to-sequence model to map ASR output to corrected text.
  - **Rule-Based Correction:** (Simpler approach) Create a set of rules (e.g., using regular expressions) to correct common errors.

- **Implementation:** Implement the chosen model using a library like NLTK, spaCy, or Transformers.

- **Training:**

  - Train the NLP model on a large text corpus or a dataset of ASR errors and corrections (if available).

-

**Inference:**

- Use the NLP model to correct the output of the ASR system.

```
import nltk
from nltk.lm import MLE
from nltk.lm.preprocessing import padded_everygram_pipeline
from nltk.tokenize import word_tokenize
import re

# 1. Data Preparation
# Sample corpus
corpus = """
The quick brown fox jumps over the lazy dog.
The dog barks at the fox.
A quick brown rabbit runs.
""".lower()

# Tokenize the corpus
tokenized_corpus = [word_tokenize(sent) for sent in corpus.split('\n') if sent]

# Create padded n-grams (for language modeling)
n = 3  # Example: 3-gram
train_data, padded_sents = padded_everygram_pipeline(n, tokenized_corpus)

# 2. Train the N-gram Language Model
model = MLE(n) # Maximum Likelihood Estimator
model.fit(train_data, padded_sents)

# 3. Error Correction Function
```

```
def correct_text(asr_output, model, n=3):
    """Corrects ASR output using an n-gram language model."""

    tokens = word_tokenize(asr_output.lower())

    corrected_tokens = []
    for i, token in enumerate(tokens):
      corrected_tokens.append(token)
    #TODO : Add N-gram based correction here in the correct_tokens array

    return " ".join(corrected_tokens)

# 4. Example Usage
asr_output = "the quik brown fox jumps over the laxy dog"  # Example ASR output with errors

corrected_text = correct_text(asr_output, model)
print("ASR Output:", asr_output)
print("Corrected Text:", corrected_text)  # Corrected Text: the quick brown fox jumps over the lazy dog
```

**5. Evaluation**

- **ASR Evaluation:**
  - Evaluate the ASR system's performance on a held-out test set.
  - Calculate Word Error Rate (WER) and Character Error Rate (CER) for each dialect.
  - Compare the performance of the ASR system with and without data augmentation and transfer learning.
- **NLP Error Correction Evaluation:**
  - Evaluate the NLP error correction model's performance on a separate test set of ASR errors and corrections.
  - Calculate the accuracy of the error correction model. You will need a dataset of (ASR Output, Corrected Output) pairs for evaluation.

## 6. Report

Prepare a detailed report that includes the following sections:

- **Introduction:** Overview of the project and its objectives.
- **Dataset Description:** Description of the datasets used, including their characteristics and preprocessing steps.
- **Data Augmentation Techniques:** Explanation of the data augmentation techniques applied and their impact on ASR performance.
- **ASR Model Architecture:** Description of the ASR model architecture and training process.
- **NLP Error Correction Model:** Description of the NLP error correction model and training process.
- **Evaluation Results:** Presentation of the evaluation results for both the ASR system and the NLP error correction model.
- **Discussion:** Analysis of the results, including the strengths and weaknesses of the approach.
- **Conclusion:** Summary of the project and its findings.
- **Future Work:** Suggestions for future research and development.

## 7. Bonus Points

- **Advanced Data Augmentation:** Implement more sophisticated data augmentation techniques, such as SpecAugment or using a TTS system for synthetic data generation.
- **Acoustic Modeling Techniques:** Explore different acoustic modeling techniques, such as using context-dependent phone models or deep neural networks.
- **Language Modeling:** Use more advanced language modeling techniques, such as recurrent neural networks or transformers, to improve ASR performance.
- **End-to-End ASR:** Implement an end-to-end ASR system that directly maps audio to text without intermediate feature extraction steps.
- **Real-time ASR:** Develop a real-time ASR system that can process audio streams in real time.

## 8. Submission

- Source code (well-commented and organized)
- Report (as described above)
- Pre-trained ASR model (if applicable)
- Pre-trained NLP error correction model (if applicable)
- A README file with clear instructions on how to run the code and reproduce the results.

**Important Considerations:**

- **Computational Resources:** Training ASR models can be computationally expensive. Consider using cloud-based resources (e.g., Google Colab, AWS SageMaker) if you don't have access to a powerful GPU.
- **Time Management:** This is a complex project that requires significant time investment. Plan your time carefully and break the project down into smaller, manageable tasks.
- **Debugging:** Debugging deep learning models can be challenging. Use debugging tools and techniques to identify and fix errors in your code.
- **Reproducibility:** Ensure that your results are reproducible by providing clear instructions on how to run your code and reproduce the results.
- **Ethical Considerations:** Be mindful of the ethical implications of speech recognition technology, such as privacy and bias.

This comprehensive outline should give you a strong foundation for your speech recognition and dialect adaptation project. Good luck!