Pronunciation Checking with ASR
Detailed Specification

Evan Nichols

MFF 2020/2021

Main Idea

The goal of this project is to present the user with a simple application that will allow them to submit recorded speech and receive a score based on their pronunciation. The application will be built using Flask, a web application framework in Python. The main computational part of the app will be using NVIDIA's NeMo toolkit, which provides the neural network module that actually grades the speech input. The application consists of a web page run in JavaScript, and a back end in Python, which uses the NN to process speech.

Interface

When looking at the webpage the user will see several elements:

- A drop-down list selection of XML library files that hold the speech prompt phrases.

- A numbered selection which pulls a phrase from the selected XML library.

- **Record** begins recording audio. The user should read the selected phrase (displayed in the text box above) aloud and press **Stop** when finished. The audio player to the right allows the user to preview their recorded audio, and the user is able to overwrite the current input by simply recording again. Upon recording, the app also creates a text file which contains the prompt. Once the recording stops, the app prepares the data set which will be used by the ASR module. This involves reformatting the audio via *ffmpeg*, and storing the audio file name, the audio's duration, and the selected phrase in a JSON file, which consists of one line that matches the following format:
  {"audio_filename": *"file.wav"*, "duration": *1.00*, "text": *"Text."*}
  This file is read by the neural network module, which creates another file with a percentage score and text keys. Also, a spectrogram of the recorded audio is displayed at the bottom of the page.

- **Grade** parses the resulting text file created by the ASR module and displays it to the user. It uses Phonemizer to show the user both regular text and the phonetic representation of their speech.

Tools & Dependencies

The application uses the latest version of Python3.
Packages and tools were installed using the **Pip** package manager for Python (version 20.3.1 at the time of writing). Some installations may fail if an older version is used.

Flask is an extensible "micro" web framework in Python. Here Flask is used to enable request dispatching between the front and back ends. It may be necessary to also use Flask-CORS (Cross Origin Resource Sharing) to bypass the security stops of some browsers. Chromium-based browsers, for example, will likely require this inclusion.
To install: *pip install Flask*
pip install -U flask-cors

NeMo is a toolkit for applications of Conversational AI. It includes extendable collections of pre-made modules and pre-trained models for automatic speech recognition, natural language processing, and text-to-speech. NeMo can utilize NVIDIA's Tensor Cores and is able to be scaled out to multiple GPUs. The current version of NeMo requires Pytorch 1.7.1, which defines the Tensor class that allows storing and operating on multidimensional rectangular arraysof numbers on CUDA-capable NVIDIA GPUs.
To install: *pip install Cython*
pip install nemo-toolkit==1.0.0b3
pip install torch torchvision torchaudio

ffmpeg is a command-line tool used to convert multimedia files between formats. In this application, the command used is *ffmpeg -i input_file.* -ar 16000 -ac 1 output.wav*. Here, *-ar 16000* specifies the sampling frequency (which is the same as the input stream by default) and *-ac 1* specifies the number of audio channels. Reformatting the audio input with this command prepares it for processing by the neural network module. This command is always run, even when it may be redundant.
To install: *pip install python-ffmpeg*

Librosa is a Python audio library, used to create a spectrogram of the user's audio input. **Matplotlib** is a Python library for creating visualizations. Here it is used to draw the spectrogram of the user's audio input.
To install: *pip install librosa*
pip install matplotlib==3.1.3

Phonemizer provides both a command-line tool and a Python function that allow one to phonemize text, which converts words and text to their corresponding phoneme tokens. Text can be converted according to the IPA standard and also to other alphabets.
To install: *pip install phonemizer*
apt-get install festival espeak-ng mbrola

Google Colab allows building Python code in modular cells, making it an excellent testing ground for this project. Colab is preferred over the similar Jupyter Notebooks in this case because Colab offers free GPU access via the CUDA package.

Details on Structure

Flask Routes

There are four main request routes which are mapped to:

- **/Library** handles all requests related to accessing the dataset(s) of phrases. Depending on the request parameter, this route will call functions that accesses the datasets in the 'lib' folder, count the phrases in a file, and return specific phrases by ID.

- **/Store_Phrase** is called once recording ends. Here a file 'sample.txt' is created and the currently selected phrase is written to it. This file resides in the folder 'dataset' and it is used later when 'dataset.json' is created, specifically the "TEXT" section.

- **/Store_Audio** stores the user's recorded speech input as 'input.wav' (the data is in sent as 'request.data', which is written to 'input.wav'). This new file is then reformatted as 'sample.wav', and the new file name, the audio's duration, and the selected phrase from 'sample.txt' are passed to **prep_Dataset()** which creates the JSON file that will be read by the ASR model. This function also calls Spectro(), which creates a spectrogram from 'input.wav'. This spectrogram is then encoded and sent back as 'response', which is displayed to the user via HTML.

- **Grade** checks for the file holding the ASR models results. If it finds the file it then parses and displays its contents to the user, or else it displays "FILE NOT FOUND."

Notable Functions

- **phrase_Get(), phrase_Num(), and getDatasets()** are the functions that act on the folder 'lib', more specifically with the .XML files inside. These functions use the *xml.etree.ElementTree* library to represent an .XML file as a tree, which enables easy searching. Phrases are found by searching the tree for the selected phrase's ID.

- **Spectro()** creates a spectrogram from the user's recorded speech. The function uses the *librosa* and *matplotlib* libraries to plot the figure and draw it. The PNG file is saved as '/temp/spectro.png.' The spectrogram is a heatmap graph, with the y-axis representing the audio frequency (Hz) and the x-axis represents the time.

- **audio_Reformat()** uses the ffmpeg command to format the audio input correctly for the ASR module. The flag **-y** forces any existing 'sample.wav' to be overwritten. The flags **-ar 16000** and **-ac 1** set the sample rate to 16000 Hz and set the audio channel count to 1, respectively.

- **ASR_Grade()** takes the 'dataset.json' file and runs it through the ASR module. The first element of the dataset is the filename of the audio ('sample.wav'), the second is that file's duration, and the third is the text key (the selected phrase). The module processes the audio, converts it to text, and compares that with the text key. The output is the calculated Word Error Rate (WER), which is rounded to a percentage and then subtracted from 100. This is so the user is shown (as a percentage) a score that represents their success, not their error.

Details on Neural Network Model

This application uses NVIDIA's NeMo toolkit. Specifically, the application uses an instance of a QuartzNet convolutional neural network. This model is instantiated using the **EncDecCTCModel** class. QuartzNet is itself a version of NVIDIA's Jasper ASR network. Quartznet makes use of separable convolutions and larger filters, enabling similar performance with fewer parameters. Like other automatic speech recognition systems, the main result is the Word Error Rate (WER). The goal of the network is to bring the WER as low as possible, to get the best possible comparison results.

The neural network makes use of a GPU for computing, which is accessed via the CUDA package. In **Grade()** there are the lines *try: from torch.cuda.amp import autocast* and *can_gpu = torch.cuda.is_available()*. Removing these lines, as well as later references to CUDA (such as *x.cuda()*) allows the module to run on just a CPU, but it will be slow and is not guaranteed to finish. GPU access was found in Google Colab and the Artificial Intelligence Cluster grid at MFF.

The 360-hour "clean" speech dataset from the LibriSpeech ASR corpus was used to train the ASR model.

Development Timeline

- Learn about ASR models and neural networks.
- Learn about NeMo specifically.
- Test ASR functions on Google Colab notebook.
- Implement XML searching.
- Assemble phrase libraries.
- Shift from Colab notebook to local python app.
- Design webpage using HTML.
- Implement audio recording and save-to-file.
- Implement Spectrogram creation w/ Librosa.
- Get application to be fully functional locally
- Shift application to UK AIC server.
- Include ASR in program, check it has GPU access.
- Sort out permissions so that all functions work from server.
- Add Unique Universally Unique Identifiers (UUIDs) to all temporary files.
- Test full functionality.

Development Details

The two main pieces of the program are implemented separately, that is: the *app.py* which powers the webpage and the *asr.py* which reads dataset files and creates "grade" files. These both have their own *requirements.txt* to remain consistent with this separation.

A python virtual environment setup was used to keep the project compact and self-contained. All packages are installed in this environment and the app files are run while it is active. The environment is created via the command: *python -m venv *directory name**, and entered using *source *directory name*/env/bin/activate*. All necessary packages can be found in the file *requirements.txt* and these can be easily downloaded using the command *pip install -r requirements.txt* while the virtual environment is active.

The addition of Universally Unique Identifiers (UUIDs) to all the temporary files the application creates was to enable multiple users to use the web page at once without having their files overwriting each other. Multiple instances running at once would result in one *temp* directory with only one of each type of file, possibly created by many different users. To stop this each instance generates an ID and should only access files with that ID prepended to their names. The spectrogram file *spectro.png* should be safer than the other files because the program creates the image file, reads it as bytes, and sends those bytes back to the web page all in sequence, but it is possible that two (or more) users hitting "Stop" simultaneously could still have their images overwritten. Just to be safe (and consistent) all temporary files have their instance's ID.

Future Work

- Add a TTS tool that allows the user to here how TTS would pronounce the prompt. This would be useful if the user encounters words with which they are not familiar. This could possibly be done with **pyttsx3** or **gTTS API**.

- Add phrases in different languages, and the necessary support from the ASR. A small menu could let users switch the UI to other languages. Phrases can be added by simply enlarging the selection of *dataset.xml* files.

- Allow users to enter their own phrases with a text-entry box. This could be done with a basic HMTL *Textbox*. Possible issues may arise from users entering non-standard characters, but this could just be prohibited to ensure at least the main functionality.

- Allow the user to choose how their audio input is represented visually. For example, they could choose to see a wave plot instead of a spectrogram.