Week 3: A speech recognition module for pilot assistance

Artificial Intelligence for Aerospace Engineering

AE-2224II - Q3 2022/3

Bianca Giovanardi

March 2, 2023

This document contains the instructions for the laboratory session of week 3. Please read it page by page, working on the exercises as you go. This laboratory session will guide you into the development of a speech recognition module. In-flight voice recognition systems allow pilots to use voice commands to perform simple cockpit tasks such as changing altitude, speed and heading. In addition to saving time in completing such tasks, another benefit of a voice-controlled cockpit is that of better situational awareness for the pilot (a pilot can keep his or her eyes on what is going on outside the cockpit, rather than diverting them to the instrument panel to activate the controls).

You will find the files necessary to complete this laboratory in directory `week3/speech-recognition` of your docker container. From now on, all the paths will be relative to that directory.

Data: Our raw data is borrowed from the TensorFlow Speech Recognition Challenge and consists of one-second audio files, containing a single spoken word (e.g. 'one', 'two', 'three', 'go', ...). These words are pronounced by a variety of different speakers. You can find these audio files in the directory `data/raw_data`, organized in subdirectories based on the word that they contain. The name of the directory therefore gives us the *label* of the audio file. Play some of the audio files directly within the VSCode environment (just click on the file) to get a sense of how the data looks like!

Aim: The aim of this laboratory session is to use the data described above to train a classifier that will 'listen' to unseen one-second recordings and recognize the word pronounced.

Part 0: Preprocessing

Let us start by processing the raw data in order to generate the dataset that will be fed into the machine learning model. To this aim, you will use `librosa`, a Python package for audio analysis. This preprocessing part of the laboratory includes the following steps:

- Step 0. Familiarize with the raw data.
- Step 1. Extract the relevant information from the raw data and build the dataset.
- Step 2. Clean up the dataset.
- Step 3. Split the dataset into the training set, the test set, and the validation set.
- Step 4. Normalize the samples in the dataset.

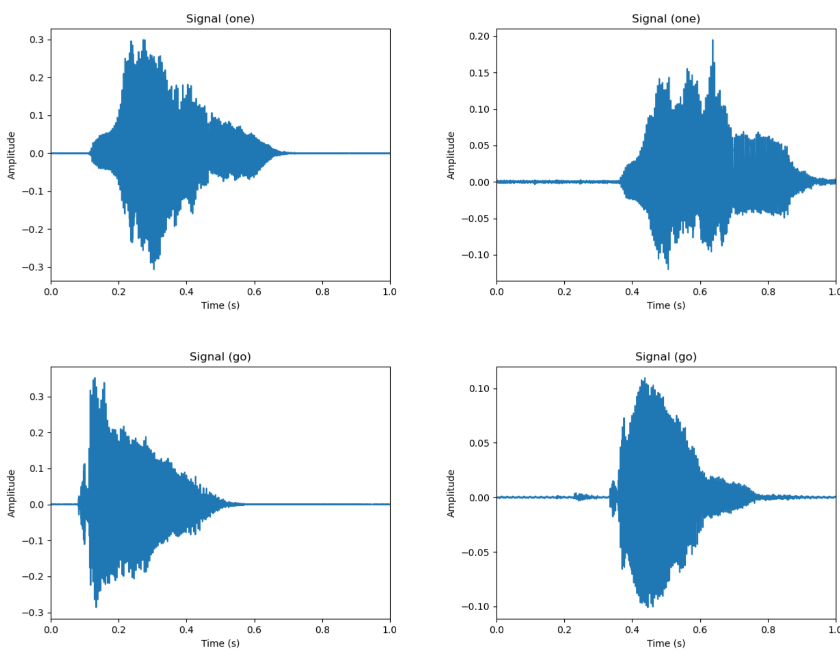
Step 0: Familiarize with the raw data

Sound is a signal, that is a variation of a quantity (in this case air pressure) over time. A signal can be represented digitally by sampling its amplitude over selected time instants. The Python package librosa allows us to extract and plot this sampling from the audio files in `data/raw_data`.

Open file `plot.py` and scroll down until the main function. You can plot the signal associated with a given sample with the function `plot_sample_signal`. For example, the following line

```
plot_sample_signal("one", sample=0)
```

will plot the signal for sample number `0` of the data labelled with `"one"` (note that the plots generated with this command will be placed in directory `plot`). Visualize the signals associated to different samples that have the same labels. For example, the figure below shows the signals associated to the words `"one"` and `"go"` pronounced by two different speakers.



Step 1: Extract features from raw data

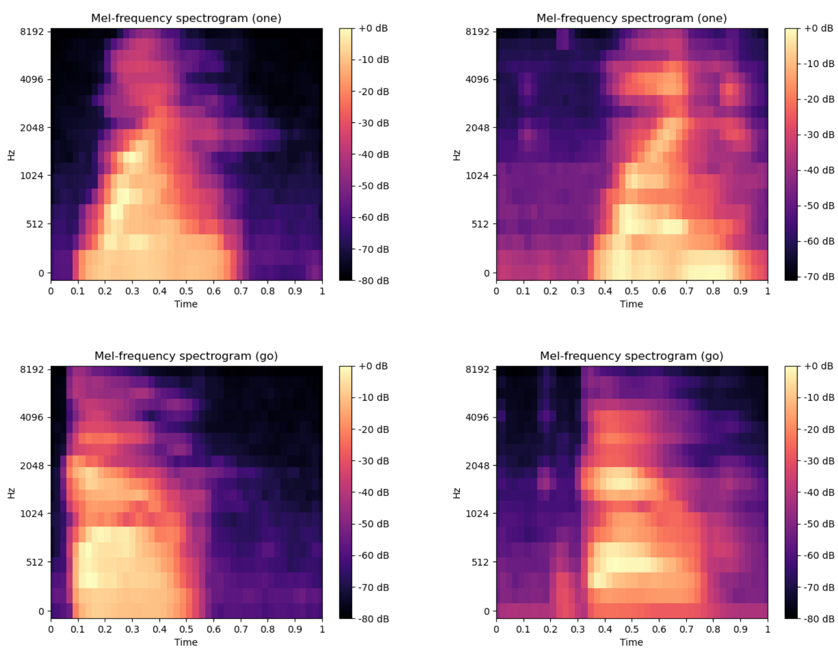
We are now going we build a set of features characterizing each of these audio files. To do so, we will transform the signal from the *time* domain to the *frequency* domain and build the *Mel spectrogram* of the audio signal.

Specifically, we will apply the Fourier Transform to the signal to identify the different frequencies that compose the audio signal (the *spectrum*). Because the signal changes over time, we will apply the Fourier transform to several short, overlapping time windows as opposed to the whole 1-second window. This procedure is implemented in `librosa.feature.melspectrogram`, see function `preprocess_file` in file `preprocess.py` for an example of how to use `librosa` to produce a Mel spectrogram. The first output of function `preprocess_file` is a 2D numpy array representing the Mel spectrogram: each entry represents the amplitude in decibels corresponding to a different band of frequencies (rows) and a different time window (columns). This 2D array represents the *features* that we will use to characterize our audio ssamples.

You can plot the Mel spectrogram associated with a given sample with the function `plot_sample_spectrogram`. (note that plots will be placed in directory `plot`). For example, the following line

```
plot_sample_spectrogram("one", sample=0)
```

will plot the spectrogram for sample number `0` of the data labelled with `"one"`. Visualize the spectrograms associated to different samples with the same labels. For example, the figure below shows the spectrograms associated to the signals analyzed above.



Because the preprocessing of all the audio samples is a quite computationally intensive task, all the samples have already been preprocessed for you. You can find the preprocessed (labeled) data in directory `data/dataset`. You can load the dataset (either in full, or a portion of it) with the following lines

```
# load whole dataset
[X, y] = load_dataset()
```

to load the whole dataset, or

```python
# load a selection of labels {labels}, {n_files_per_label} files per label
input_labels = ["one", "two", "three"]
n_files_per_label = 1000
[X, y] = load_dataset_selection(input_labels, n_files_per_label)
```

to load a subset of the dataset (selected labels and given number of samples per label). Both functions return two `pandas` `DataFrames`:

X: the *data* (each row contains the flattened Mel spectrograms of an audio sample loaded)

y: the *label* (each row contains the string with the corresponding word)

Step 2: Data cleaning

Open file `clean_split_scale.py` and scroll down until the main function. This file loads a quite small selection of the dataset (3 labels and 10 files per label) and prints the dataset `X`. Run the file. Observe that quite some samples in your dataset exhibit `NaN`s in some of the spectrogram entries. In the interest of time, we will not analyze where do these `NaN`s come from, but rather we will remove these samples from our dataset altogether.

Implement in file `clean_split_scale.py` the following function:

```python
def clean(X, y):
    """
    TODO:
    Part 0, Step 2:
        - Use the pandas {isna} and {dropna} functions to remove
            from the dataset any corrupted samples
    """

    # return the cleaned data
    return [X, y]
```

to remove the `NaN` samples from `X`.

Hints:

- Remember to remove the corresponding labels from `y` too!
- Check out the pandas documentation of isna and dropna.

Step 3: Data splitting in train, test, and validation sets

You will then split the dataset into three subsets, namely the *training* set, the *test* set, and the *validation* set. Each of these subsets will serve a different purpose in the following parts of the laboratory. You will use:

- the *training* set to train the machine learning algorithm;
- the *test* set to assess the performance of the machine learning algorithm;
- the *validation* set to fine tune the model parameters.

Implement in file `clean_split_scale.py` the following function:

```python
def train_test_validation_split(X, y, test_size, cv_size):

    """
    TODO:
    Part 0, Step 3:
        - Use the sklearn {train_test_split} function to split the dataset
            (and the labels) into train, test and validation sets
    """

    # return split data
    return [X_train, y_train, X_test, y_test, X_cv, y_cv]
```

that splits the dataset into the train, test, and validation set. The input arguments `test_size` and `cv_size` (between 0 and 1) represent the desired ratios of the test and validation sets, respectively, with respect to the whole dataset loaded.

Hint:

- Check out the sklearn documentation of function train_test_split. Shuffle the dataset (`shuffle=True`) and set the random state to zero (`random_state=0`), so we all consistently get the same (reproducible) results for the same inputs.

Step 4: Data normalization

Finally, do not forget to normalize the dataset!

To this aim, implement in file `clean_split_scale.py` the following function:

```python
def scale(X_train, X_test, X_cv):

    """
    TODO:
    Part 0, Step 4:
        - Use the {preprocessing.StandardScaler} of sklearn to normalize the data
        - Scale the train, test and validation sets accordingly
    """

    # return the scaler
    return [X_train, X_test, X_cv, scaler]
```

Hint:

- Check out the sklearn documentation of the StandardScaler class.

Now that you have completed the implementation of the tree functions `clean`, `train_test_validation_split`, and `scale`, you should be able to call correctly the following function:

```python
# cleanup data, split data in training set and test set, normalize data
[X_train, y_train, X_test, y_test, X_cv, y_cv, scaler] = clean_split_scale(X, y)
```

which does the cleaning/splitting/scaling in one pass, by calling the three functions that you have implemented. From now on, use `test_size = 0.1` and `cv_size = 0.1`.

Submit your answers to Questions 0.2, 0.3 and 0.4 in weblab.

Part 1: Binary Classification with SVM

For this part, you will edit the files `main_binary_svm.py` and `train.py`. For simplicity, we start by considering a dataset with only two words – `"one"` and `"two"` – and 100 samples per label.

1. Implement the function `train_binary_svm_classifier` in file `train.py`. This function should take in input the training subset of the data `X_train`, the associated labels `y_train`, the regularization parameter `C` and the kernel coefficient `gamma`, and return a trained Support Vector Machine (SVM) classifier `clf` using a radial basis function kernel:

   ```python
   def train_binary_svm_classifier(X_train, y_train, C, gamma):
       """
       Train a binary Support Vector Machine classifier with sk-learn
       """

       """
       TODO:
       Part 1:
           - Use the sklearn {svm.SVC} class to implement a binary classifier
       """

       return clf
   ```

   Hint: Check out the sklearn documentation of class SVC.

2. Complete the main function in `main_binary_svm.py`, so as to call function `train_binary_svm_classifier` to train the binary SVM classifier with `C = 10` and `gamma = 0.001`.

3. Use function `evaluate` (which is implemented in `evaluate.py`) to assess the accuracy of the classifier on the test subset of the dataset in terms of precision, recall and F1-score.

4. Use function `display_confusion_matrix` (which is implemented in `evaluate.py`) to display the *confusion matrix* of the trained classifier (note that the plots generated with this command will be placed in directory `plot`).

5. Increment the number of samples per label to `1000` and rerun the analysis. Comment on the result.

Submit your answers to Part 1 in weblab.

Part 2: Multi-class Classification with SVM

For this part, you will edit the files main_multi_class_svm.py and train.py. Now consider a dataset with four words – "one", "two", "three", and "go" – and 1000 samples per label.

1. Implement the function train_multi_class_svm_classifier in file train.py. This function should take in input the training subset of the data X_train, the associated labels y_train, the regularization parameter C and the kernel coefficient gamma, and return a trained Support Vector Machine (SVM) classifier clf using a radial basis function kernel:

   ```python
   def train_multi_class_svm_classifier(X_train, y_train, C, gamma):
       """
       Train a multi-class Support Vector Machine classifier with sk-learn
       """

       """
       TODO:
       Part 2:
           - Use the sklearn {OneVsRestClassifier} class to implement a
               multi-class classifier
       """

       return clf
   ```

   Hint: Check out the sklearn documentation of class OneVsRestClassifier.

2. Use the validation portion of the dataset to select an appropriate value for gamma. Specifically, on the same plot visualize the F1-score obtained on the training set and on the validation set with the following values of gamma: [1e-5, 1e-4, 1e-3, 1e-2, 1e-1]. Set C = 10 and use a logarithmic scale for the x-axis. Comment on the result. What is the best value for gamma?

3. Train the SVM classifier with C = 10 and the optimal value for gamma obtained from the validation. Use function evaluate to assess the accuracy of classifier on the test subset of the dataset in terms of precision, recall and F1-score.

Submit your answers to Part 2 in weblab.

The remaining of the assignment is OPTIONAL.

OPTIONAL - Part 3: Multi-class Classification with Neural Networks

For this part, you will edit the files `main_multi_class_NN.py` and `MLPClassifier_torch.py` (it is also recommended to take a look at function `train_multi_class_nn_classifier` in `train.py`). Now consider a dataset with four words – `"one"`, `"two"`, `"three"`, and `"go"` – and 1000 samples per label.

1. Implement a neural network classifier with a cross-entropy loss function in pytorch. Use one hidden layer of size 30 with ReLU activation, the Adam optimizer with `weight_decay=1.e-4` and `lr=1.e-3`.

   Hint: Check out the pytorch documentation of class CrossEntropyLoss use the torch.nn.Softmax function to extract the numeric label prediction.

   NOTE: Check out how functions `encode_array` and `decode_array` (implemented in `utils.py`) are used to transform the string labels into numeric labels and viceversa. Then, check out how function torch.nn.functional.one_hot is used to transform numeric labels into arrays of size equal to the number of classes and value of each component equal to `0` or `1` depending on whether that component matches the label.

2. Train the neural network on the training set and evaluate its performance on the test set.

3. From now on consider all the labels in the dataset and 1000 samples per label. Train the same neural network used above on the new training set and evaluate its performance on the test set. What do you observe?

4. Plot the F1-score on the training and validation sets for the following number of units in the hidden layer: 30, 50, 100, and 150. Use this plot to chose an appropriate size of the hidden layer.

5. Plot the F1-score on the training and validation sets for an increasing number of samples per label: 100, 200, 400, 600, 800, and 1000. Do you think that collecting more data will help improve the performance of your neural network?