Speech Recognition System

Cluster Innovation Centre
University of Delhi

Submitted to **Prof. Shobha Bagai** and **Dr. Parveen** in partial fulfilment for the degree of
B.Tech. (Information Technology and Mathematical Innovations)

for paper
V.6.2 Signals and Systems

14/11/2018

By:
Vaibhav Jain
B.Tech. (Information Technology and Mathematical Innovations)
[2016]

Table of Contents

INTRODUCTION

Speech is the most basic means of adult human communication. The basic goal of speech processing is to provide an interaction between a human and a machine.

Far from a being a fad, the overwhelming success of speech-enabled products like Amazon Alexa has proven that some degree of speech support will be an essential aspect of household tech for the foreseeable future. If you think about it, the reasons why are pretty obvious.

The accessibility improvements alone are worth considering. Speech recognition allows the elderly and the physically and visually impaired to interact with state-of-the-art products and services quickly and naturally—no GUI needed!

Speech processing system has mainly three tasks −

- First, speech recognition that allows the machine to catch the words, phrases and sentences we speak
- Second, natural language processing to allow the machine to understand what we speak, and
- Third, speech synthesis to allow the machine to speak.

In this report, we focus on speech recognition, the process of understanding the words that are spoken by human beings. Remember that the speech signals are captured with the help of a microphone and then it has to be understood by the system.

How Speech Recognition Works – An Overview

Speech recognition has its roots in research done at Bell Labs in the early 1950s. Early systems were limited to a single speaker and had limited vocabularies of about a dozen words. Modern speech recognition systems have come a long way since their ancient counterparts. They can recognize speech from multiple speakers and have enormous vocabularies in numerous languages.

The first component of speech recognition is, of course, speech. Speech must be converted from physical sound to an electrical signal with a microphone, and then to digital data with an analog-to-digital converter. Once digitized, several models can be used to transcribe the audio to text.

Most modern speech recognition systems rely on what is known as a Hidden Markov Model (HMM). This approach works on the assumption that a speech signal, when viewed on a short enough time scale (say, ten milliseconds), can be reasonably approximated as a stationary process—that is, a process in which statistical properties do not change over time.

In a typical HMM, the speech signal is divided into 10-millisecond fragments. The power spectrum of each fragment, which is essentially a plot of the signal's power as a function of frequency, is mapped to a vector of real numbers known as cepstral coefficients. The dimension of this vector is usually small—sometimes as low as 10, although more accurate systems may have dimension 32 or more. The final output of the HMM is a sequence of these vectors.

To decode the speech into text, groups of vectors are matched to one or more phonemes—a fundamental unit of speech. This calculation requires training, since the sound of a phoneme varies from speaker to speaker, and even varies from one utterance to another by the same speaker. A special algorithm is then applied to determine the most likely word (or words) that produce the given sequence of phonemes.

One can imagine that this whole process may be computationally expensive. In many modern speech recognition systems, neural networks are used to simplify the speech signal using techniques for feature transformation and dimensionality reduction before HMM recognition.

Building a Speech Recognizer

Speech Recognition or Automatic Speech Recognition (ASR) is the center of attention for AI projects like robotics. Without ASR, it is not possible to imagine a cognitive robot interacting with a human. However, it is not quite easy to build a speech recognizer.

Difficulties in developing a speech recognition system

Developing a high quality speech recognition system is really a difficult problem. The difficulty of speech recognition technology can be broadly characterized along a number of dimensions as discussed below −

- Size of the vocabulary − Size of the vocabulary impacts the ease of developing an ASR. Consider the following sizes of vocabulary for a better understanding.
    - A small size vocabulary consists of 2-100 words, for example, as in a voice-menu system
    - A medium size vocabulary consists of several 100s to 1,000s of words, for example, as in a database-retrieval task
    - A large size vocabulary consists of several 10,000s of words, as in a general dictation task.

  Note that, the larger the size of vocabulary, the harder it is to perform recognition.

- Channel characteristics − Channel quality is also an important dimension. For example, human speech contains high bandwidth with full frequency range, while a telephone speech consists of low bandwidth with limited frequency range. Note that it is harder in the latter.
- Speaking mode − Ease of developing an ASR also depends on the speaking mode, that is whether the speech is in isolated word mode, or connected word mode, or in a continuous speech mode. Note that a continuous speech is harder to recognize.

- Speaking style − A read speech may be in a formal style, or spontaneous and conversational with casual style. The latter is harder to recognize.

- Speaker dependency − Speech can be speaker dependent, speaker adaptive, or speaker independent. A speaker independent is the hardest to build.

- Type of noise − Noise is another factor to consider while developing an ASR. Signal to noise ratio may be in various ranges, depending on the acoustic environment that observes less versus more background noise −
    - If the signal to noise ratio is greater than 30dB, it is considered as high range
    - If the signal to noise ratio lies between 30dB to 10db, it is considered as medium SNR
    - If the signal to noise ratio is lesser than 10dB, it is considered as low range

  For example, the type of background noise such as stationary, non-human noise, background speech and crosstalk by other speakers also contributes to the difficulty of the problem.

- Microphone characteristics − The quality of microphone may be good, average, or below average. Also, the distance between mouth and microphone can vary. These factors also should be considered for recognition systems.

Despite these difficulties, researchers worked a lot on various aspects of speech such as understanding the speech signal, the speaker, and identifying the accents.

Rest of this report is intended as a step-by-step tutorial to build your own Automatic Speech Recognition (ASR) system. You will have to follow the steps given below to build a speech recognizer

Visualizing Audio Signals - Reading from a File and Working on it

This is the first step in building speech recognition system as it gives an understanding of how an audio signal is structured. Some common steps that can be followed to work with audio signals are as follows −

Recording

Before doing any work, the first step is to record the audio which want to work with. Speech audio can be easily recorded using a microphone, but the quality of the audio is the biggest factor.

Sampling

When recording with microphone, the signals are stored in a digitized form. But to work upon it, the machine needs them in the discrete numeric form. Hence, we should perform sampling at a certain frequency and convert the signal into the discrete numerical form. Choosing the high frequency for sampling implies that when humans listen to the signal, they feel it as a continuous audio signal.

Programming Example: Analysing Audio Signals

The following example shows a stepwise approach to analyze an audio signal, using Python, which is stored in a file. The frequency of this audio signal is 44,100 HZ.

Import the necessary packages as shown here −

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
```

Now, read the stored audio file. It will return two values: the sampling frequency and the audio signal. Provide the path of the audio file where it is stored, as shown here −

```
frequency_sampling, audio_signal = wavfile.read("a2002011001-e02.wav")
```

Next, display the parameters like sampling frequency of the audio signal, data type of signal and its duration, using the commands shown −

```
print('\nSignal shape:', audio_signal.shape)
print('Signal Datatype:', audio_signal.dtype)
print('Signal duration:', round(audio_signal.shape[0] /
float(frequency_sampling), 2), 'seconds')
```

This step involves normalizing the signal as shown below −

```
audio_signal = audio_signal / np.power(2, 15)
```

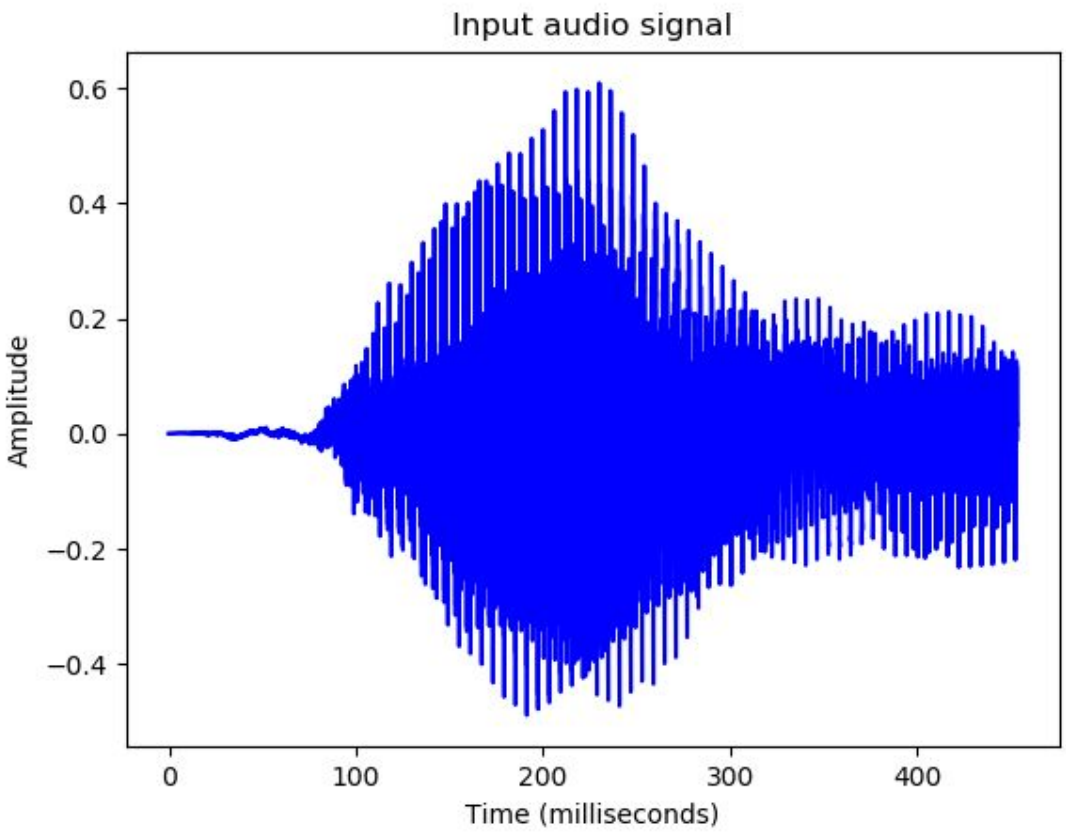
In this step, we are extracting the first 100 values from this signal to visualize. Use the following commands for this purpose −

```
audio_signal = audio_signal [:5000]
time_axis = 1000 * np.arange(0, len(signal), 1) / float(frequency_sampling)
```

Now, visualize the signal using the commands given below −

```
plt.plot(time_axis, signal, color='blue')
plt.xlabel('Time (milliseconds)')
plt.ylabel('Amplitude')
plt.title('Input audio signal')
plt.show()
```

You would be able to see an output graph and data extracted for the above audio signal as shown in the image here -




```
Signal shape: (2395137,2)
Signal Datatype: int16
Signal duration: 54.31 seconds
```

Characterizing the Audio Signal: Transforming to Frequency Domain

Characterizing an audio signal involves converting the time domain signal into frequency domain, and understanding its frequency components, by. This is an important step because it gives a lot of information about the signal. You can use a mathematical tool like Fourier Transform to perform this transformation.

Programming Example: Transforming to Frequency Domain using Fourier Transformation

The following example shows, step-by-step, how to characterize the signal, using Python, which is stored in a file. Note that here we are using Fourier Transform mathematical tool to convert it into frequency domain.

Import the necessary packages, as shown here −

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
```

Now, read the stored audio file. It will return two values: the sampling frequency and the the audio signal. Provide the path of the audio file where it is stored as shown in the command here −

```
frequency_sampling, audio_signal = wavfile.read("a2002011001-e02.wav")
```

In this step, we will display the parameters like sampling frequency of the audio signal, data type of signal and its duration, using the commands given below −

```
print('\nSignal shape:', audio_signal.shape)
print('Signal Datatype:', audio_signal.dtype)
print('Signal duration:', round(audio_signal.shape[0] /
float(frequency_sampling), 2), 'seconds')
```

In this step, we need to normalize the signal, as shown in the following command−

```
audio_signal = audio_signal / np.power(2, 15)
```

This step involves extracting the length and half length of the signal. Use the following commands for this purpose −

```
length_signal = len(audio_signal)
half_length = np.ceil((length_signal + 1) / 2.0).astype(np.int)
```

Now, we need to apply mathematics tools for transforming into frequency domain. Here we are using the Fourier Transform.

```
signal_frequency = np.fft.fft(audio_signal)
```

Now, do the normalization of frequency domain signal and square it −

```
signal_frequency = abs(signal_frequency[0:half_length]) / length_signal
signal_frequency **= 2
```

Next, extract the length and half length of the frequency transformed signal −

```
len_fts = len(signal_frequency)
```

Note that the Fourier transformed signal must be adjusted for even as well as odd case.

```
if length_signal % 2:
   signal_frequency[1:len_fts] *= 2
else:
   signal_frequency[1:len_fts-1] *= 2
```

Now, extract the power in decibel (dB) −

```
signal_power = 10 * np.log10(signal_frequency)
```

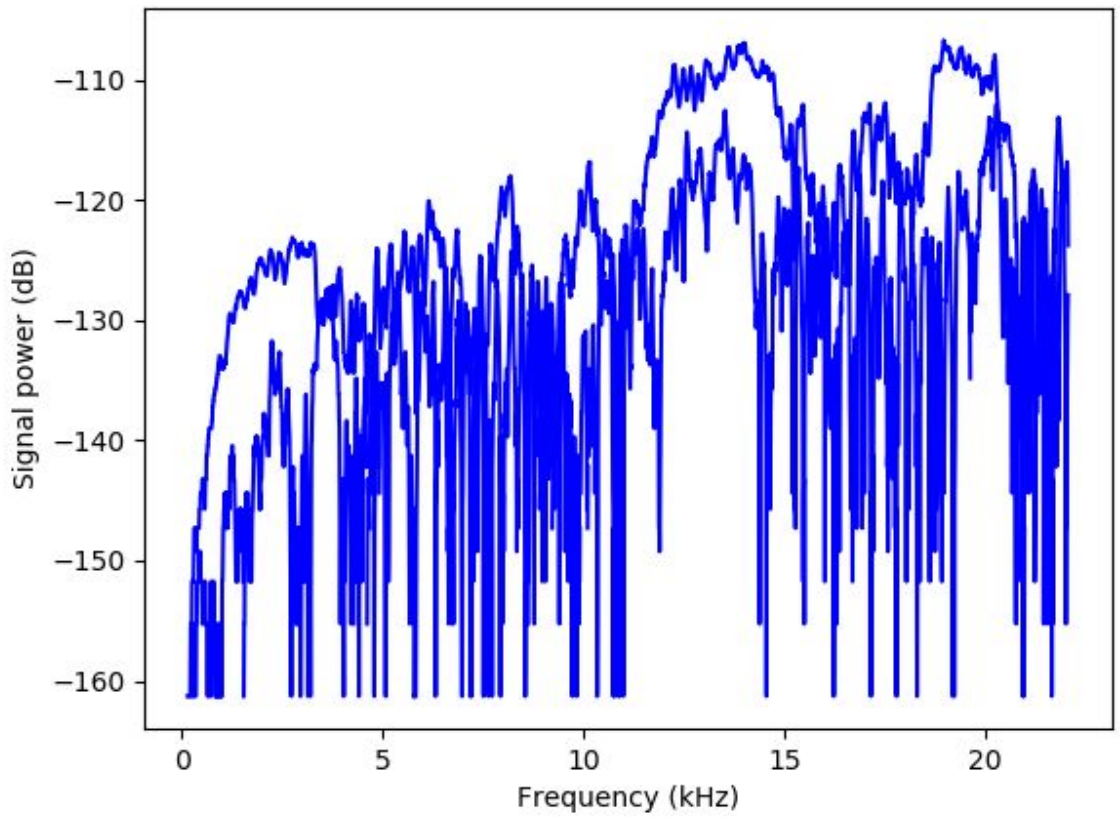
Adjust the frequency in kHz for X-axis −

```
x_axis = np.arange(0, len_half, 1) * (frequency_sampling / length_signal) / 1000.0
```

Now, visualize the characterization of signal as follows −

```
plt.figure()
plt.plot(x_axis, signal_power, color='blue')
plt.xlabel('Frequency (kHz)')
plt.ylabel('Signal power (dB)')
plt.show()
```

You can observe the output graph of the above code as shown in the image below −



Generating Monotone Audio Signal

The two steps that you have seen till now are important to learn about signals. Now, this step will be useful if you want to generate the audio signal with some predefined parameters. Note that this step will save the audio signal in an output file.

Programming Example: Generating Monotone Audio Signals

In the following example, we are going to generate a monotone signal, using Python, which will be stored in a file. For this, you will have to take the following steps −

Import the necessary packages as shown −

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import write
```

Provide the file where the output file should be saved

```
output_file = 'audio_signal_generated.wav'
```

Now, specify the parameters of your choice, as shown −

```
duration = 4 # in seconds
frequency_sampling = 44100 # in Hz
frequency_tone = 784
min_val = -4 * np.pi
max_val = 4 * np.pi
```

In this step, we can generate the audio signal, as shown −

```
t = np.linspace(min_val, max_val, duration * frequency_sampling)
audio_signal = np.sin(2 * np.pi * tone_freq * t)
```

Now, save the audio file in the output file −

```
write(output_file, frequency_sampling, signal_scaled)
```

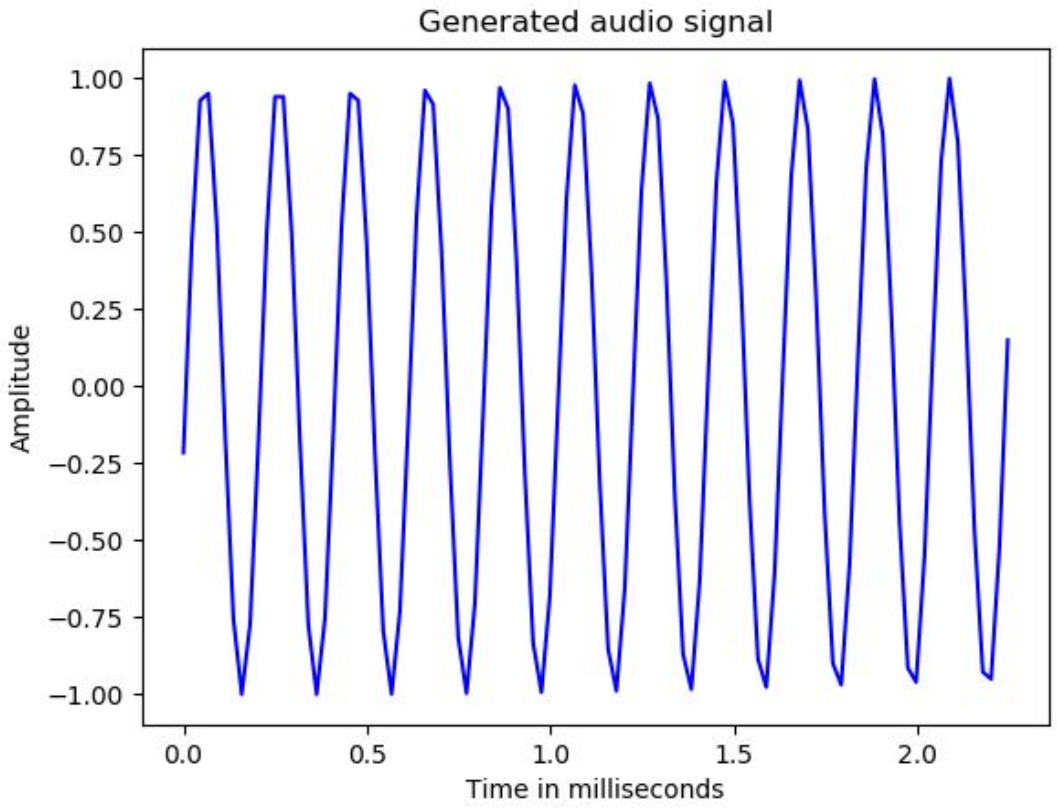
Extract the first 100 values for our graph, as shown –

```
audio_signal = audio_signal[:100]
time_axis = 1000 * np.arange(0, len(signal), 1) / float(sampling_freq)
```

Now, visualize the generated audio signal as follows –

```
plt.plot(time_axis, signal, color='blue')
plt.xlabel('Time in milliseconds')
plt.ylabel('Amplitude')
plt.title('Generated audio signal')
plt.show()
```

You can observe the plot as shown in the figure given here –



Feature Extraction from Speech

This is the most important step in building a speech recognizer because after converting the speech signal into the frequency domain, we must convert it into the usable form of feature vector. We can use different feature extraction techniques like MFCC, PLP, PLP-RASTA etc. for this purpose.

Programming Example: Feature Extraction from Speech Signal

In the following example, we are going to extract the features from signal, step-by-step, using Python, by using MFCC technique.

Import the necessary packages, as shown here −

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from python_speech_features import mfcc, logfbank
```

Now, read the stored audio file. It will return two values – the sampling frequency and the audio signal. Provide the path of the audio file where it is stored.

```
frequency_sampling, audio_signal = wavfile.read("a2002011001-e02.wav")
```

Note that here we are taking first 15000 samples for analysis.

```
audio_signal = audio_signal[:15000]
```

Use the MFCC techniques and execute the following command to extract the MFCC features –

```
features_mfcc = mfcc(audio_signal, frequency_sampling)
```

Now, print the MFCC parameters, as shown –

```
print('\nMFCC:\nNumber of windows =', features_mfcc.shape[0])
print('Length of each feature =', features_mfcc.shape[1])
```

Now, plot and visualize the MFCC features using the commands given below –

```
features_mfcc = features_mfcc.T
plt.matshow(features_mfcc)
plt.title('MFCC')
```

In this step, we work with the filter bank features as shown –
Extract the filter bank features –

```
filterbank_features = logfbank(audio_signal, frequency_sampling)
```

Now, print the filterbank parameters.

```
print('\nFilter bank:\nNumber of windows =', filterbank_features.shape[0])
print('Length of each feature =', filterbank_features.shape[1])
```

Now, plot and visualize the filterbank features.

```
filterbank_features = filterbank_features.T
plt.matshow(filterbank_features)
plt.title('Filter bank')
plt.show()
```

As a result of the steps above, you can observe the following outputs: Figure1 for MFCC and Figure2 for Filter Bank

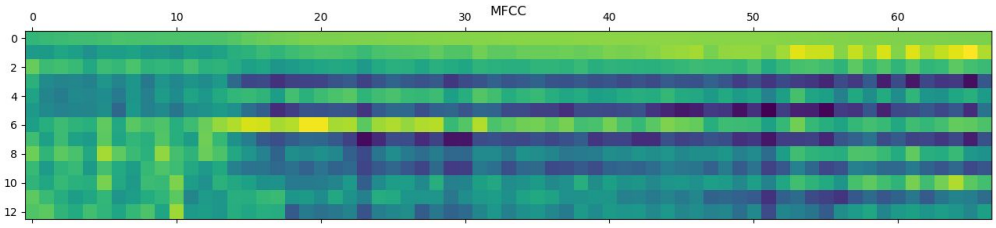


Figure 1

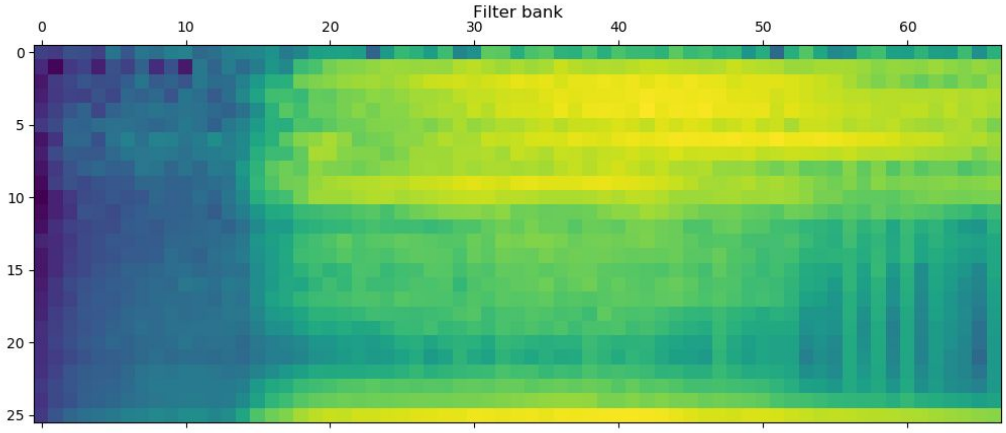


Figure 2

Recognition of Spoken Words

Speech recognition means that when humans are speaking, a machine understands it. Fortunately, you don't have to worry about any of this. A number of speech recognition services are available for use online through an API, and many of these services offer Python SDKs.

Picking a Python Speech Recognition Package

A handful of packages for speech recognition exist on PyPI. A few of them include:

- ➔ apiai
- ➔ assemblyai
- ➔ google-cloud-speech
- ➔ pocketsphinx
- ➔ SpeechRecognition
- ➔ watson-developer-cloud
- ➔ wit

Some of these packages—such as wit and apiai—offer built-in features, like natural language processing for identifying a speaker's intent, which go beyond basic speech recognition. Others, like google-cloud-speech, focus solely on speech-to-text conversion.

There is one package that stands out in terms of ease-of-use: SpeechRecognition.

Recognizing speech requires audio input, and SpeechRecognition makes retrieving this input really easy. Instead of having to build scripts for accessing microphones and processing audio files from scratch, SpeechRecognition will have you up and running in just a few minutes.

The SpeechRecognition library acts as a wrapper for several popular speech APIs and is thus extremely flexible. One of these—the Google Web Speech API—supports a default API key that is hard-coded into the SpeechRecognition library. That means you can get off your feet without having to sign up for a service.

The flexibility and ease-of-use of the SpeechRecognition package make it an excellent choice for any Python project. However, support for every feature of each API it wraps is not guaranteed. You will need to spend some time researching the available options to find out if SpeechRecognition will work in your particular case.

So, now that you're convinced you should try out SpeechRecognition, the next step is getting it installed in your environment.

Installing SpeechRecognition

You can install SpeechRecognition from a terminal with pip:

```
$ pip install SpeechRecognition
```

Once installed, you should verify the installation by opening an interpreter session and typing:

```
>>> import speech_recognition as sr
>>> sr.__version__
'3.8.1'
```

SpeechRecognition *will* work out of the box if all you need to do is work with existing audio files. Specific use cases, however, require a few dependencies. Notably, the PyAudio package is needed for capturing microphone input.

The Recognizer Class

All of the magic in SpeechRecognition happens with the Recognizer class.

The primary purpose of a Recognizer instance is, of course, to recognize speech. Each instance comes with a variety of settings and functionality for recognizing speech from an audio source.

Creating a Recognizer instance is easy. In your current interpreter session, just type:

```
r = sr.Recognizer()
```

Each Recognizer instance has seven methods for recognizing speech from an audio source using various APIs. These are:

1. recognize_bing(): Microsoft Bing Speech
2. recognize_google(): Google Web Speech API
3. recognize_google_cloud(): Google Cloud Speech - requires installation of the google-cloud-speech package
4. recognize_houndify(): Houndify by SoundHound
5. recognize_ibm(): IBM Speech to Text
6. recognize_sphinx(): CMU Sphinx - requires installing PocketSphinx
7. recognize_wit(): Wit.ai

Of the seven, only recognize_sphinx() works offline with the CMU Sphinx engine. The other six all require an internet connection.

A full discussion of the features and benefits of each API is beyond the scope of this tutorial. Since SpeechRecognition ships with a default API key for the Google Web Speech API, you can get started with it right away.

Each recognize_*() method will throw a speech_recognition.RequestError exception if the API is unreachable. For recognize_sphinx(), this could happen as the result of a missing, corrupt or incompatible Sphinx installation. For the other six methods, RequestError may be thrown if quota limits are met, the server is unavailable, or there is no internet connection.

Programming Example: Creating a Speech Recognizer System

Import the necessary packages as shown −

```
import speech_recognition as sr
```

Create an object as shown below −

```
recording = sr.Recognizer()
```

Now, instead of using an audio file as the source, we will use the default system microphone. We can access this by creating an instance of the Microphone class. If

your system has no default microphone (such as on a RaspberryPi), or you want to use a microphone other than the default, you will need to specify which one to use by supplying a device index.

You can capture input from the microphone using the listen() method of the Recognizer class inside of the with block. This method takes an audio source as its first argument and records input from the source until silence is detected.

To handle ambient noise, you'll need to use the adjust_for_ambient_noise() method of the Recognizer class. Since input from a microphone is far less predictable than input from an audio file, it is a good idea to do this anytime you listen for microphone input.

```
with sr.Microphone() as source: recording.adjust_for_ambient_noise(source)
   print("Please Say something:")
   audio = recording.listen(source)
```

Now google API would recognize the voice and gives the output.

```
try:
   print("You said: \n" + recording.recognize_google(audio))
except Exception as e:
   print(e)
```

Once you execute the program, try speaking "hello" into your microphone. Wait a moment for the interpreter prompt to display again. Now you're ready to recognize the speech.

You can see the following output −

```
Please Say Something:
You said:
```

For example, if you said **"I am BATMAN"**, then the system recognizes it correctly as follows −

```
I am BATMAN
```

Recap and Conclusion

In this tutorial, we have learned about Speech Recognition Systems. We discussed about its advantages and disadvantages, technical problem faced in practice and etc. We've seen how to install the SpeechRecognition package and use its Recognizer class to easily recognize speech from both a file—using record()—and microphone input—using listen(). We also saw how to process segments of an audio file using the offset and duration keyword arguments of the record() method.

We've seen the effect noise can have on the accuracy of transcriptions, and have learned how to adjust a Recognizer instance's sensitivity to ambient noise with adjust_for_ambient_noise().

Speech recognition is a deep subject, and what we have learned here barely scratches the surface. I hope that this document have served as a stepping stone in journey in the field of Signal Processing and Speech Recognition.

References

[1] https://realpython.com/python-speech-recognition/#working-with-microphones

[2] https://github.com/Uberi/speech_recognition/blob/master/reference/library-reference.rst

[3] https://github.com/Uberi/speech_recognition/tree/master/examples

[4] https://www.youtube.com/watch?v=yxxRAHVtafI&feature=youtu.be

[5] https://cacm.acm.org/magazines/ 2014/1/170863-a-historical-perspective-of-speech-recognition/abstract

[6] https://medium.com/swlh/the-past-present-and-future-of-speech-recognition-technology-cf13c179aaf

[7] http://www.iitg.ac.in/samudravijaya/tutorials/fundamentalOfASR_picone96.pdf

[8] https://mitpress.mit.edu/books/voice-machine

[9] https://www.tutorialspoint.com/artificial_intelligence_with_python/artificial_intelligence _with _python_speech_recognition.htm