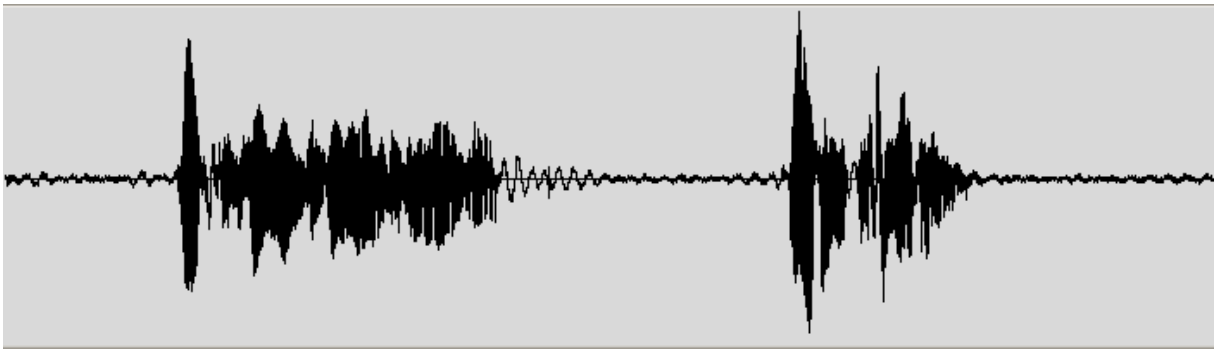
Basic concepts of speech recognition

Speech is a complex phenomenon. People rarely understand how is it produced and perceived. The naive perception is often that speech is built with words and each word consists of phones. The reality is unfortunately very different. Speech is a dynamic process without clearly distinguished parts. It's always useful to get a sound editor and look into the recording of the speech and listen to it. Here is for example the speech recording in an audio editor.



All modern descriptions of speech are to some degree probabilistic. That means that there are no certain boundaries between units, or between words. Speech to text translation and other applications of speech are never 100% correct. That idea is rather unusual for software developers, who usually work with deterministic systems. And it creates a lot of issues specific only to speech technology.

Structure of speech

In current practice, speech structure is understood as follows:

Speech is a continuous audio stream where rather stable states mix with dynamically changed states. In this sequence of states, one can define more or less similar classes of sounds, or **phones**. Words are understood to be built of phones, but this is certainly not true. The acoustic properties of a waveform corresponding to a phone can vary greatly depending on many factors - phone context, speaker, style of speech and so on. The so-called coarticulation makes phones sound very different from their "canonical" representation. Next, since transitions between words are more informative than stable regions, developers often talk about **diphones** - parts of phones between two consecutive phones. Sometimes developers talk about subphonetic units - different substates of a phone. Often three or more regions of a different nature can be found.

The number three can easily be explained: The first part of the phone depends on its preceding phone, the middle part is stable and the next part depends on the subsequent phone. That's why there are often three states in a phone selected for speech recognition.

Sometimes phones are considered in context. Such phones in context are called **triphones** or even **quinphones**. For example "u" with left phone "b" and right phone "d" in the word "bad" sounds a bit different than the same phone "u" with left phone "b" and right phone "n" in word "ban". Please note that unlike diphones, they are matched with the same range in waveform as just phones. They just differ by name because they describe slightly different sounds.

For computational purpose it is helpful to detect parts of triphones instead of triphones as a whole, for example if you want to create a detector for the beginning of a triphone and share it across many triphones. The whole variety of sound detectors can be represented by a small amount of distinct short sound detectors. Usually we use 4000 distinct short sound detectors to compose detectors for triphones. We call those detectors **senones**. A senone's dependence on context can be more complex than just the left and right context. It can be a rather complex function defined by a decision tree, or in some other ways.

Next, phones build subword units, like syllables. Sometimes, syllables are defined as "reduction-stable entities". For instance, when speech becomes fast, phones often change, but syllables remain the same. Also, syllables are related to an intonational contour. There are other ways to build subwords - morphologically-based (in morphology-rich languages) or phonetically-based. Subwords are often used in open vocabulary speech recognition.

Subwords form words. Words are important in speech recognition because they restrict combinations of phones significantly. If there are 40 phones and an average word has 7 phones, there must be 40^7 words. Luckily, even people with a rich vocabulary rarely use more then 20k words in practice, which makes recognition way more feasible.

Words and other non-linguistic sounds, which we call **fillers** (breath, um, uh, cough), form **utterances**. They are separate chunks of audio between pauses. They don't necessary match sentences, which are more semantic concepts.

On the top of this, there are dialog acts like turns, but they go beyond the purpose of this document.

Recognition process

The common way to recognize speech is the following: we take a waveform, split it at utterances by silences and then try to recognize what's being said in each utterance. To do that, we want to take all possible combinations of words and try to match them with the audio. We choose the best matching combination.

There are some important concepts in this matching process. First of all it's the concept of **features**. Since the number of parameters is large, we are trying to optimize it. Numbers that are calculated from speech usually by dividing the speech into frames. Then for each frame, typically of 10 milliseconds length, we extract 39 numbers that represent the speech. That's called a **feature vector**. The way to generate the number of parameters is a subject of active investigation, but in a simple case it's a derivative from the spectrum.

Second, it's the concept of the **model**. A model describes some mathematical object that gathers common attributes of the spoken word. In practice, for an audio model of senone it is the gaussian mixture of it's three states - to put it simple, it's the most probable feature vector. From the concept of the model the following issues raise:

- how well does the model describe reality,
- can the model be made better of it's internal model problems and
- how adaptive is the model if conditions change

The model of speech is called Hidden Markov Model or HMM. It's a generic model that describes a black-box communication channel. In this model process is described as a sequence of states which change each other with a certain probability. This model is intended to describe any sequential process like speech. HMMs have been proven to be really practical for speech decoding.

Third, it's a matching process itself. Since it would take longer than universe existed to compare all feature vectors with all models, the search is often optimized by applying many tricks. At any points we maintain the best matching variants and extend them as time goes on, producing the best matching variants for the next frame.

Models

According to the speech structure, three models are used in speech recognition to do the match:

An **acoustic model** contains acoustic properties for each senone. There are context-independent models that contain properties (the most probable feature vectors for each phone) and context-dependent ones (built from senones with context).

A **phonetic dictionary** contains a mapping from words to phones. This mapping is not very effective. For example, only two to three pronunciation variants are noted in it. However, it's practical enough most of the time. The dictionary is not the only method for mapping words to phones. You could also use some complex function learned with a machine learning algorithm.

A **language model** is used to restrict word search. It defines which word could follow previously recognized words (remember that matching is a sequential process) and helps to significantly restrict the matching process by stripping words that are not probable. The most common language models are n-gram language models–these contain statistics of word sequences–and finite state language models–these define speech sequences by finite state automation, sometimes with weights. To reach a good accuracy rate, your language model must be very successful in search space restriction. This means it should be very good at predicting the next word. A language model usually restricts the vocabulary that is considered to the words it contains. That's an issue for name recognition. To deal with this, a language model can contain smaller chunks like subwords or even phones. Please note that the search space restriction in this case is usually worse and the corresponding recognition accuracies are lower than with a word-based language model.

Those three entities are combined together in an engine to recognize speech. If you are going to apply your engine for some other language, you need to get such structures in place. For

many languages there are acoustic models, phonetic dictionaries and even large vocabulary language models available for download.

Other used concepts

A **Lattice** is a directed graph that represents variants of the recognition. Often, getting the best match is not practical. In that case, lattices are good intermediate formats to represent the recognition result.

N-best lists of variants are like lattices, though their representations are not as dense as the lattice ones.

Word confusion networks (sausages) are lattices where the strict order of nodes is taken from lattice edges.

Speech database - a set of typical recordings from the task database. If we develop dialog system it might be dialogs recorded from users. For dictation system it might be reading recordings. Speech databases are used to train, tune and test the decoding systems.

Text databases - sample texts collected for e.g. language model training. Usually, databases of texts are collected in sample text form. The issue with such a collection is to put present documents (like PDFs, web pages, scans) into a spoken text form. That is, you need to remove tags and headings, to expand numbers to their spoken form and to expand abbreviations.

What is optimized

When speech recognition is being developed, the most complex problem is to make search precise (consider as many variants to match as possible) and to make it fast enough to not run for ages. Since models aren't perfect, another challenge is to make the model match the speech.

Usually the system is tested on a test database that is meant to represent the target task correctly.

The following characteristics are used:

Word error rate. Let's assume we have an original text and a recognition text with a length of *N*words. *I* is the number of inserted words, *D* is the number of deleted words and *S* represent the number of substituted words. With this, the word error rate can be calculated as

$$WER = (I + D + S) / N$$

The WER is usually measured in percent.

Accuracy. It is almost the same as the word error rate, but it doesn't take insertions into account.

$$Accuracy = (N - D - S) / N$$

For most tasks, the accuracy is a worse measure than the WER, since insertions are also important in the final results. However, for some tasks, the accuracy is a reasonable measure of the decoder performance.

Speed. Suppose an audio file has a recording time (RT) of 2 hours and the decoding took 6 hours. Then the speed is counted as 3xRT.

ROC curves. When we talk about detection tasks, there are false alarms and hits/misses. To illustrate these, ROC curves are used. Such a curve is a diagram that describes the number of false alarms versus the number of hits. It tries to find the optimal point where the number of false alarms is small and the number of hits matches 100%.

There are other properties that aren't often taken into account, but still important for many practical applications. Your first task should be to build such a measure and systematically apply it during the system development. Your second task is to collect the test database and test how your application performs.

Overview of the CMUSphinx toolkit

The CMUSphinx toolkit is a leading speech recognition toolkit with various tools used to build speech applications. CMUSphinx contains a number of packages for different tasks and applications. Sometimes, it's confusing what to choose. To shed some light on the parts of the toolkit, here is a list:

- Pocketsphinx — lightweight recognizer library written in C.
- Sphinxbase — support library required by Pocketsphinx
- Sphinx4 — adjustable, modifiable recognizer written in Java
- Sphinxtrain — acoustic model training tools

We recommend that you use the latest available releases:

- sphinxbase-5prealpha
- pocketsphinx-5prealpha
- sphinx4-5prealpha
- sphinxtrain-5prealpha

Of course, many things are missing. Things like building a phonetic model capable of handling an infinite vocabulary, postprocessing of the decoding result, sense extraction and other semantic tools should be added one day. Probably you should take it on.

The following resources are the main ones for CMUSphinx developers:

- Website
- Forum
- Mailing list
- Download page
- Github
- Telegram Chat

Before you start

Before you start developing a speech application, you need to consider several important points. They will define the way you will implement your application.

Algorithms

Speech technology sets several important limits to the way you implement an application. For example, as noted before, it is *impossible* to recognize *any* known word of the language. You need to consider ways to overcome such limitations. Such ways are known for most types of applications out there and will be described later in the tutorial. To follow them, you sometimes need to rethink how your application will behave and interact with the user.

Although we try to provide important examples, we obviously can't cover everything. There is no utterance verification or speaker identification example yet, though they could be created later. Most algorithms are widely covered in scientific literature and some of them are explained later in the tutorial. Moreover, new methods to solve old problems raise each year.

Let's go through a list of several common applications and how to approach them:

Generic dictation is never as generic as its name says. You need to determine a domain to recognize which can be e.g. dialogs, readings, meetings, voicemails, legal or medical transcriptions. If you consider voicemails, note that the language in this domain is way more restricted than general language. It's actually a very small vocabulary with specialized sequences of terms:

- It's Sandy. Let's meet tomorrow
- Hi. That's Joe, I'm going to sell you that car

There will be a lot of names, which is certainly a problem, but you'll never find a voicemail about quantum physics – and that's a very good thing. The recognizer will use the restrictions you provided with the language model in order to improve the accuracy of the result.

You'll have to build a language model for your domain, but that's not as complicated as you might think. It doesn't matter, if you only covered the 60k most common words in English; the accuracy will be the same as if you considered 120k words. For other languages with rich morphology the situation is different, but also solvable with morphology-based subwords. Also, you have to build a post-processing system, an adaptation system and a user-identification system.

For **recognition on an embedded processor,** there are two ways to consider – recognition on the server and recognition on the device. The former is more popular nowadays because it leverages the power and flexibility of cloud computing.

Language learning will require you to build a framework for tracking incorrect pronunciations. That will include the generation and scoring of incorrect pronunciations.

For **command and control,** it was popular to use a finite state grammar for a long time. However, we do not recommend this approach nowadays. It's way better to employ a medium vocabulary recognizer with a semantic analysis framework on top to improve the users' experience and let them use more or less natural language. In short, don't build command and control, build intelligent assistants instead.

For **intelligent assistants** you do not only need the recognition, but also intent parsing and database knowledge. For more details on how to implement this you can check Lucida powered byOpenEphyra. Dialog systems will require a framework for user feedback as well.

Voice search, semantic analysis and translation will need to be built on top of the lattices generated by an engine. You need to take lattices with confidence scores and feed them into the upper levels like a translation engine.

For open vocabulary recognition like **name and places recognition,** you will need a subword language model.

Text alignment, like captions synchronization, will require you to build a specialized language model from a reference text to restrict the search.

Existing accuracy results

For most tasks above there are published accuracy results. You can look into them if you identify the task. Those results can or cannot be useful in terms of accuracy for your users. I might be that your accuracy results appear to be better than the ones from the publications. However, this might be unlikely and might not be as easy to achieve as it seems at first.

Let's for instance consider a broadcast news recognition system which has an accuracy of 20-25%. If this is not enough for your application, you probably need to consider modifying the application. You might add a manual correction step or a preliminary adaptation step to improve accuracy. If the accuracy will not be sufficient afterwards, then it's probably better to think about whether you need speech at all. There are other, more reliable, interfaces you could use.

For instance, though ASR-based IVR systems are fancy and handy, many people still prefer communication with DTMF systems or web-based forms or just email to contact the company. Remember that you need an effective interface, not a modest one.

Resources

Another problem you need to consider is the availability of speech material for training, testing and optimizing the system. You need to find out which resources are available to you.

The **testing set** is a critical issue for any speech recognition application. The testing set should be representative enough acoustically and in terms of the language. On the other hand, the test set doesn't necessarily need be large, you can spend ten minutes to create a good one. It might be a couple of recordings you could do yourself.

For the training set and the models you should check the resources that are already present. The increasing interest in speech technology makes people contribute by creating models for their native languages. In general, you'll have to collect audio material for a specified language. Actually it's not that complicated. Audio books, movies and podcasts provide enough recordings to build a very good acoustic model with little effort.

To build a phonetic dictionary you can use one of the existing TTS synthesizers which nowadays cover a lot of languages. You can also boostrap a dictionary by hand and then extend it with machine learning tools.

For language models you'll have to find a lot of texts for your domain. It might be textbooks, already transcribed recordings or some other sources like website contents crawled on the web.

Technologies

A third thing to consider is the set of particular technologies you will build on. Although CMUSphinx tries to provide a more or less complete programming suite for developing speech applications, you'll sometimes need to use other packages/programming languages/tools. You need to find out yourself if you need to continue with Java, C or any of the scripting languages CMUSphinx supports. A simple rule to choose between sphinx4 and pocketsphinx is the following:

- If you need *speed or portability* ⇢ use pocketsphinx
- If you need *flexibility and managability* ⇢ use sphinx4

Although people often ask whether sphinx4 or pocketsphinx is more accurate, you shouldn't bother with this question at all. Accuracy is *not* the argument here. Both sphinx4 and pocketsphinx provide sufficient accuracy and even then it depends on many factors, not just the engine. The point is that the engine is just a part of the system which should include many more components. If we are talking about a large vocabulary decoder, there must be a diarization framework, an adaptation framework and a postprocessing framework. They all need to cooperate somehow. The flexibility of sphinx4 allows you to build such a system quickly. It's easy to embed sphinx4 into a flash server like red5 to provide web-based recognition. It's easy to manage many sphinx4 instances doing large-scale decoding on a cluster.

On the other side, if your system needs to be efficient and reasonably accurate, if you are running on embedded device or if you are interested in using a recognizer with some exotic language like Erlang, then pocketsphinx is your choice. It's very hard to integrate Java with other languages that are not supported by the JVM – pocketsphinx is way better in this case.

Last, you need to choose your development platform. If you are bound to a specific one, that's an easy task for you. If you can choose, we highly recommend to use GNU/Linux as your development platform. We can help you with Windows or Mac issues but there are no

guarantees – our main development platform is Linux. For many tasks you'll need to run complex scripts using Perl or Python. On Windows it might be problematic.

Alright, let's start! The next sections will describe the process of creating a sample application with either sphinx4 or pocketsphinx. Choose the one that fits for you.

Building an application with sphinx4

Caution!
This tutorial uses the sphinx4 API from the 5 pre-alpha release.
The API described here is not supported in earlier versions.

Overview

Sphinx4 is a pure Java speech recognition library. It provides a quick and easy API to convert the speech recordings into text with the help of CMUSphinx acoustic models. It can be used on servers and in desktop applications. Besides speech recognition, Sphinx4 helps to identify speakers, to adapt models, to align existing transcription to audio for timestamping and more.

Sphinx4 supports US English and many other languages.

Using sphinx4 in your projects

As any library in Java all you need to do to use sphinx4 is to add the jars to the dependencies of your project and then you can write code using the API.

The easiest way to use sphinx4 is to use modern build tools like Apache Maven or Gradle. Sphinx-4 is available as a maven package in the Sonatype OSS repository.

In gradle you need the following lines in `build.gradle`:

```
repositories {

    mavenLocal()

    maven { url "https://oss.sonatype.org/content/repositories/snapshots" }
```

```
}



dependencies {

    compile group: 'edu.cmu.sphinx', name: 'sphinx4-core', version:'5prealpha-
SNAPSHOT'

    compile group: 'edu.cmu.sphinx', name: 'sphinx4-data', version:'5prealpha-
SNAPSHOT'

}
```

To use sphinx4 in your maven project specify this repository in your `pom.xml`:

```
<project>

...

    <repositories>

        <repository>

            <id>snapshots-repo</id>

            <url>https://oss.sonatype.org/content/repositories/snapshots</url>

            <releases>

                <enabled>false</enabled>

            </releases>

            <snapshots>

                <enabled>true</enabled>

            </snapshots>

        </repository>

    </repositories>

...

</project>
```

Then add `sphinx4-core` to the project dependencies:

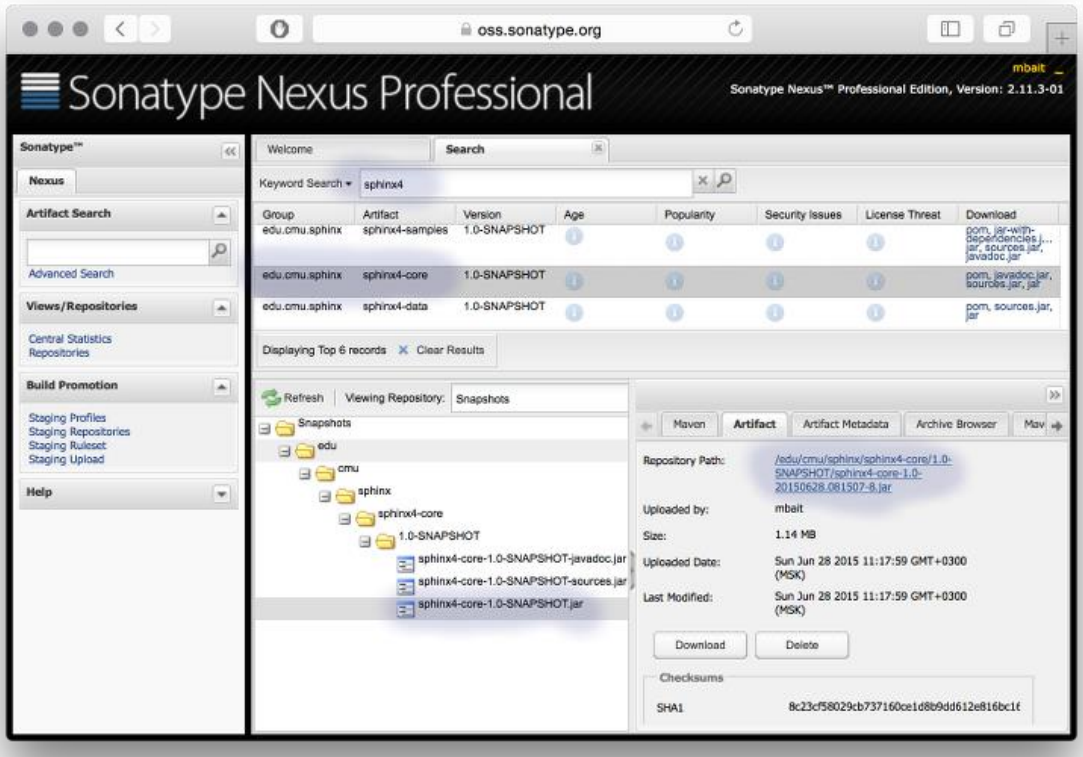
```
<dependency>

  <groupId>edu.cmu.sphinx</groupId>

  <artifactId>sphinx4-core</artifactId>

  <version>5prealpha-SNAPSHOT</version>

</dependency>
```

Add `sphinx4-data` to the dependencies as well if you want to use the default US English acoustic and language models:

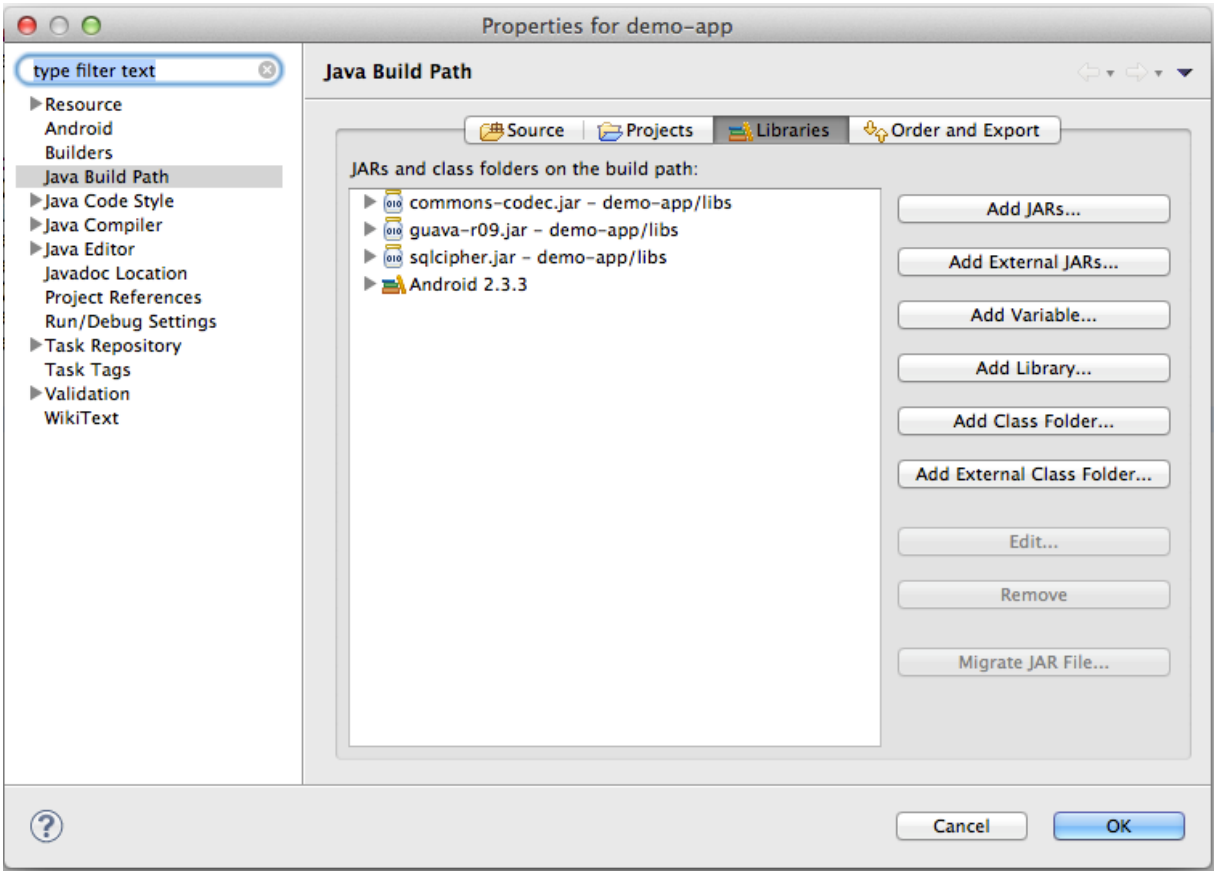
```
<dependency>

  <groupId>edu.cmu.sphinx</groupId>

  <artifactId>sphinx4-data</artifactId>

  <version>5prealpha-SNAPSHOT</version>

</dependency>
```

Many IDEs like Eclipse, Netbeans or Idea have support for Gradle either through plugins or with built-in features. In that case you can just include sphinx4 libraries into your project with the help of your IDE. Please check the relevant part of your IDE documentation, for example the IDEA documentation on Gradle.

You can also use Sphinx4 in a non-maven project. In this case you need to download the jars from the repository manually. You might also need to download the dependencies (which we try to keep small) and include them in your project. You need the *sphinx4-core* jar and the *sphinx4-data* jar if you are going to use US English acoustic model:



Here is an example for how to include the jars in Eclipse:



Basic Usage

To quickly start with sphinx4, create a java project as described above, add the required dependencies and type the following simple code:

```
package com.example;


import java.io.File;

import java.io.FileInputStream;

import java.io.InputStream;


import edu.cmu.sphinx.api.Configuration;

import edu.cmu.sphinx.api.SpeechResult;

import edu.cmu.sphinx.api.StreamSpeechRecognizer;


public class TranscriberDemo {


    public static void main(String[] args) throws Exception {
```

```
        Configuration configuration = new Configuration();


        configuration.setAcousticModelPath("resource:/edu/cmu/sphinx/models/en-
us/en-us");
        configuration.setDictionaryPath("resource:/edu/cmu/sphinx/models/en-
us/cmudict-en-us.dict");
        configuration.setLanguageModelPath("resource:/edu/cmu/sphinx/models/en-
us/en-us.lm.bin");


        StreamSpeechRecognizer recognizer = new
StreamSpeechRecognizer(configuration);
        InputStream stream = new FileInputStream(new File("test.wav"));


        recognizer.startRecognition(stream);
        SpeechResult result;
        while ((result = recognizer.getResult()) != null) {
            System.out.format("Hypothesis: %s\n", result.getHypothesis());
        }
        recognizer.stopRecognition();
    }
}
```

This simple code snippet transcribes the file `test.wav` – just make sure it exists in the project root.

There are several high-level recognition interfaces in sphinx4:

- LiveSpeechRecognizer
- StreamSpeechRecognizer
- SpeechAligner

For most of the speech recognition jobs high-level interfaces should be sufficient. Basically, you will only have to setup four attributes:

- Acoustic model
- Dictionary
- Grammar/Language model
- Source of speech

The first three attributes are set up using a `Configuration` object which is then passed to a recognizer. The way to connect to a speech source depends on your concrete recognizer and usually is passed as a method parameter.

Configuration

A `Configuration` is used to supply the required and optional attributes to the recognizer.

```
Configuration configuration = new Configuration();


// Set path to acoustic model.

configuration.setAcousticModelPath("resource:/edu/cmu/sphinx/models/en-us/en-us");

// Set path to dictionary.

configuration.setDictionaryPath("resource:/edu/cmu/sphinx/models/en-us/cmudict-en-us.dict");

// Set language model.

configuration.setLanguageModelPath("resource:/edu/cmu/sphinx/models/en-us/en-us.lm.bin");
```

LiveSpeechRecognizer

The `LiveSpeechRecognizer` uses a microphone as the speech source.

```
LiveSpeechRecognizer recognizer = new LiveSpeechRecognizer(configuration);

// Start recognition process pruning previously cached data.

recognizer.startRecognition(true);

SpeechResult result = recognizer.getResult();

// Pause recognition process. It can be resumed then with startRecognition(false).

recognizer.stopRecognition();
```

StreamSpeechRecognizer

The `StreamSpeechRecognizer` uses an `InputStream` as the speech source. You can pass the data from a file, a network socket or from an existing byte array.

```
StreamSpeechRecognizer recognizer = new StreamSpeechRecognizer(configuration);

recognizer.startRecognition(new FileInputStream("speech.wav"));


SpeechResult result = recognizer.getResult();

recognizer.stopRecognition();
```

Please note that the audio for this decoding must have one of the following formats:

```
RIFF (little-endian) data, WAVE audio, Microsoft PCM, 16 bit, mono 16000 Hz
```

or

```
RIFF (little-endian) data, WAVE audio, Microsoft PCM, 16 bit, mono 8000 Hz
```

The decoder does not support other formats. If the audio format does not match, you will not get any results. This means, you need to convert your audio to a proper format before decoding. E.g. if you want to decode audio in telephone quality with a sample rate of 8000 Hz, you would need to call

```
configuration.setSampleRate(8000);
```

You can retreive multiple results until the end of the file is reached:

```
while ((result = recognizer.getResult()) != null) {

    System.out.println(result.getHypothesis());

}
```

SpeechAligner

A `SpeechAligner` time-aligns text with audio speech.

```
SpeechAligner aligner = new SpeechAligner(configuration);

recognizer.align(new URL("101-42.wav"), "one oh one four two");
```

SpeechResult

A `SpeechResult` provides access to various parts of the recognition result, such as the recognized utterance, a list of words with timestamps, the recognition lattice, etc.:

```
// Print utterance string without filler words.

System.out.println(result.getHypothesis());


// Get individual words and their times.

for (WordResult r : result.getWords()) {

    System.out.println(r);

}


// Save lattice in a graphviz format.

result.getLattice().dumpDot("lattice.dot", "lattice");
```

Demos

A number of sample demos are included in the sphinx4 sources in order to give you an understanding how to run sphinx4. You can run them from the sphinx4-samples jar:

- Transcriber - demonstrates how to transcribe a file
- Dialog - demonstrates how to lead a dialog with a user
- SpeakerID - speaker identification
- Aligner - demonstration of audio to transcription timestamping

If you are going to start with a demo please do not modify the demo inside the sphinx4 sources. Instead, copy the code into your project and modify it there.

Building from source

If you want to develop sphinx4 itself you might want to build it from source.
Sphinx4 uses the Gradle build system. In order to compile and install everything, including the dependencies, simply type 'gradle build' in the root directory.

If you are going to use an IDE, make sure it supports Gradle projects. Then simply import the sphinx4 source tree.

Troubleshooting

You might experience the one or the other problem while using sphinx4. Please check the FAQ first before asking any new questions on the forum.

In case you have issues with the accuracy, you need to provide the audio recording you are trying to recognize along with all models you use. Additionally, you need to describe in which way your results differ from your expectations.

Building an application with PocketSphinx

Caution!
This tutorial describes pocketsphinx 5 pre-alpha release.
It is not going to work for older versions.

Installation

PocketSphinx is a library that depends on another library called SphinxBase which provides common functionality across all CMUSphinx projects. To install Pocketsphinx, you need to install both Pocketsphinx and Sphinxbase. You can use Pocketsphinx with Linux, Windows, on MacOS, iPhone and Android.

First of all, download the released packages for pocketsphinx and sphinxbase from the projects download page or check them out from Subversion or Github.

For more details see the download page.

Unpack them into the same directory. On Windows, you will need to rename 'sphinxbase-X.Y' (where X.Y is the SphinxBase version number) to simply 'sphinxbase' to satisfy the project configuration of pocketsphinx.

Installation on Unix system

To build pocketsphinx in a Unix-like environment (such as Linux, Solaris, FreeBSD etc.) you need to make sure you have the following dependencies installed: gcc, automake, autoconf, libtool, bison, swig (at least version 2.0), the Python development package and the pulseaudio development package.

If you want to build it without some dependencies you can use the respective configuration options like `--without-swig-python`. However, for beginners it's recommended to install all dependencies.

You need to download both the sphinxbase and pocketsphinx packages and unpack them. Please note that you cannot use sphinxbase and pocketsphinx of different version. So, please make sure that their versions are in sync. After unpacking, you should see the following two main folders:

```
sphinxbase-X.X

pocketsphinx-X.x
```

First, build and install SphinxBase. Change the current directory to the `sphinxbase` folder. If you downloaded it directly from the repository, you need to run the following command at least once to generate the `configure` file:

```
./autogen.sh
```

If you downloaded the release version, or ran `autogen.sh` at least once, then compile and install it with:

```
./configure

make

make install
```

The last step might require root permissions, so you might need to run `sudo make install`. If you want to use fixed-point arithmetic, you must configure SphinxBase with the `--enable-fixed`option. You can also set an installation prefix with `--prefix` or configure to use or not to use SWIG Python support.

Sphinxbase will be installed in the `/usr/local/` directory by default. Not all systems load libraries from this folder automatically. In order to load them you need to configure the path to look for shared libaries. This can be done either in the `/etc/ld.so.conf` file or by exporting the environment variables:

```
export LD_LIBRARY_PATH=/usr/local/lib

export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
```

For more details on the linker configuration see the Shared Libraries HOWTO.

Then change to the pocketsphinx folder and perform the same steps:

```
./configure

make

make install
```

To test the installation, run `pocketsphinx_continuous -inmic yes` and check that it recognizes words you speak into your microphone.

If you get an error such as: `error while loading shared libraries: libpocketsphinx.so.3`, you may want to check your linker configuration of the `LD_LIBRARY_PATH` environment variable described above.

Windows

In MS Windows, under MS Visual Studio 2012 (or newer – we test with Visual C++ 2012 Express):

- load `sphinxbase.sln` in the sphinxbase directory
- compile all the projects in SphinxBase (from `sphinxbase.sln`)
- load `pocketsphinx.sln` in the pocketsphinx directory
- compile all the projects in PocketSphinx

MS Visual Studio will build the executables and libraries under `.\bin\Release` or `.\bin\Debug`(depending on the target you choose on MS Visual Studio). In order to run `pocketsphinx_continuous.exe`, don't forget to copy the sphinxbase.dll file to the bin folder. Otherwise the executable will fail to find this library. Unlike on Linux, the path to the model is not preconfigured in Windows, so you have to specify for pocketsphinx_continuous where to find the model with the `-hmm`, `-lm` and `-dict` options.

To recognize speech from your microphone, change to the pocketsphinx folder and run:

```
bin\Release\Win32\pocketsphinx_continuous.exe -inmic yes

    -hmm model\en-us\en-us -lm model\en-us\en-us.lm.bin

    -dict model\en-us\cmudict-en-us.dict
```

To recognize speech from a file run:

```
bin\Release\Win32\pocketsphinx_continuous.exe -infile test\data\goforward.raw

    -hmm model\en-us\en-us -lm model\en-us\en-us.lm.bin

    -dict model\en-us\cmudict-en-us.dict
```

Pocketsphinx API core ideas

The Pocketsphinx API is designed to ease the use of speech recognizer functionality in your applications:

1. It is very likely to remain stable both in terms of source and binary compatibility, due to the use of abstract types.
2. It is fully re-entrant, so there is no problem having multiple decoders in the same process.
3. It allows a drastic reduction in code footprint and a modest but significant reduction in memory consumption.

The reference documentation for the new API is available at https://cmusphinx.github.io/doc/pocketsphinx/.

Basic usage (hello world)

There are a few key things you need to know when you want to use the API:

1. command-line parsing is done externally (in `<cmd_ln.h>`)
2. everything takes a `ps_decoder_t *` as the first argument

To illustrate the API, we will step through a simple “hello world” example. This example is somewhat specific to Unix regarding the locations of files and the compilation process. We will create a C source file called `hello_ps.c`. To compile it (on Unix), use this command:

```
gcc -o hello_ps hello_ps.c \

    -DMODELDIR=\"`pkg-config --variable=modeldir pocketsphinx`\" \

    `pkg-config --cflags --libs pocketsphinx sphinxbase`
```

Please note that compilation errors mean that you didn’t carefully follow the installation guide above. For example pocketsphinx needs to be properly installed to be available through the pkg-config system. To check whether pocketsphinx is installed properly, just run

```
pkg-config --cflags --libs pocketsphinx sphinxbase
```

on the command line and make sure the output looks like this:

```
-I/usr/local/include -I/usr/local/include/sphinxbase -
I/usr/local/include/pocketsphinx

        -L/usr/local/lib -lpocketsphinx -lsphinxbase -lsphinxad
```

Initialization

The first thing we need to do is to create a configuration object, which for historical reasons is called `cmd_ln_t`. Along with the general boilerplate for our C program, our code looks like this:

```
#include <pocketsphinx.h>


int
main(int argc, char *argv[])
{
    ps_decoder_t *ps = NULL;
    cmd_ln_t *config = NULL;

    config = cmd_ln_init(NULL, ps_args(), TRUE,
                         "-hmm", MODELDIR "/en-us/en-us",
                         "-lm", MODELDIR "/en-us/en-us.lm.bin",
                         "-dict", MODELDIR "/en-us/cmudict-en-us.dict",
                         NULL);

    return 0;
}
```

The `cmd_ln_init()` function takes a variable number of null-terminated string arguments, followed by NULL. The first argument is any previous `cmd_ln_t *` which is to be updated. The second argument is an array of argument definitions – the standard set can be obtained by calling`ps_args()`. The third argument is a flag telling the argument parser to be “strict”. If this argument is `TRUE`, then duplicate arguments or unknown arguments will cause the parsing process to fail.

The `MODELDIR` macro is defined on the GCC command-line by using the `pkg-config` to obtain the `modeldir` variable from the PocketSphinx configuration. On Windows, you can simply add a preprocessor definition to the code, such as this:

```
#define MODELDIR "c:/sphinx/model"
```

(replace this with wherever your models are installed). In order to initialize the decoder, use `ps_init`:

```
ps = ps_init(config);
```

Decoding a file stream

Because live audio input is somewhat platform-specific, we will confine ourselves to decoding audio files. There is an audio file helpfully included in the pocketsphinx source code which contains this very sentence. You can find it in `pocketsphinx/test/data/goforward.raw`. Copy it to the current directory. If you want to create your own version of it: it needs to be a single-channel (monaural), little-endian, unheadered 16-bit signed PCM audio file sampled at 16000 Hz.

The main pocketsphinx use case is to read audio data in blocks of memory from a source and feed it to the decoder. To do that, we first open the file and start decoding the utterance using `ps_start_utt()`:

```
rv = ps_start_utt(ps);
```

Next, we read 512 samples at a time from the file, and feed them to the decoder using `ps_process_raw()`:

```
int16 buf[512];

while (!feof(fh)) {

    size_t nsamp;

    nsamp = fread(buf, 2, 512, fh);

    ps_process_raw(ps, buf, nsamp, FALSE, FALSE);

}
```

Afterwards, we need to mark the end of the utterance using `ps_end_utt()`:

```
    rv = ps_end_utt(ps);
```

Last, we retrieve the hypothesis to get our recognition result:

```
    hyp = ps_get_hyp(ps, &score);

    printf("Recognized: %s\n", hyp);
```

We can also retrieve the hypothesis during recognition. If we do so, it will return a partial result.

Cleaning up

To clean up, simply call `ps_free()` on the object that was returned by `ps_init()`. Free the configuration object with `cmd_ln_free_r`.

Full code listing

Here is the full listing of our code again:

```
#include <pocketsphinx.h>
```

```
int
main(int argc, char *argv[])
{
    ps_decoder_t *ps;
    cmd_ln_t *config;
    FILE *fh;
    char const *hyp, *uttid;
    int16 buf[512];
    int rv;
    int32 score;

    config = cmd_ln_init(NULL, ps_args(), TRUE,
                         "-hmm", MODELDIR "/en-us/en-us",
                         "-lm", MODELDIR "/en-us/en-us.lm.bin",
                         "-dict", MODELDIR "/en-us/cmudict-en-us.dict",
                         NULL);
    if (config == NULL) {
        fprintf(stderr, "Failed to create config object, see log for details\n");
        return -1;
    }

    ps = ps_init(config);
    if (ps == NULL) {
        fprintf(stderr, "Failed to create recognizer, see log for details\n");
        return -1;
    }

    fh = fopen("goforward.raw", "rb");
    if (fh == NULL) {
        fprintf(stderr, "Unable to open input file goforward.raw\n");
        return -1;
```

```
    }

    rv = ps_start_utt(ps);

    while (!feof(fh)) {
        size_t nsamp;
        nsamp = fread(buf, 2, 512, fh);
        rv = ps_process_raw(ps, buf, nsamp, FALSE, FALSE);
    }

    rv = ps_end_utt(ps);
    hyp = ps_get_hyp(ps, &score);
    printf("Recognized: %s\n", hyp);

    fclose(fh);
    ps_free(ps);
    cmd_ln_free_r(config);

    return 0;
}
```

Advanced usage

For more advanced uses of the API please check the API reference.

- For word segmentations, the API provides an iterator object which is used to iterate over the sequence of words. This iterator object is an abstract type, with some accessors provided to obtain timepoints, scores and, most interestingly, posterior probabilities for each word.
- The confidence of the whole utterance can be accessed with the `ps_get_prob` method.
- You can access the lattice if needed.
- You can configure multiple searches and switch between them in runtime.

Searches

As a developer you can configure several “search” objects with different grammars and language models and switch between them during runtime to provide interactive experience for the user.

There are multiple possible search modes:

- *keyword*: efficiently looks for a keyphrase and ignores other speech. It Allows to configure the detection threshold.
- *grammar*: recognizes speech according to the JSGF grammar. Unlike keyphrase search, grammar search doesn't ignore words which are not in the grammar but tries to recognize them.
- *ngram/lm*: recognizes natural speech with a language model.
- *allphone*: recognizes phonemes with a phonetic language model.

Each search has a name and can be referenced by a name. Names are application-specific. The function `ps_set_search` allows to activate the search that was previously added by a name.

In order to add a search, one needs to point to the grammar/language model describing the search. The location of the grammar is specific to the application. If only a simple recognition is required it is sufficient to add a single search or to just configure the required mode using configuration options.

The exact design of a search depends on your application. For example, you might want to listen for an activation keyword first and once this keyword is recognized switch to ngram search to recognize the actual command. Once you recognized the command you can switch to grammar search to recognize the confirmation and then switch back to keyword listening mode to wait for another command.

Building a phonetic dictionary

Introduction

A phonetic dictionary provides the system with a mapping of vocabulary words to sequences of phonemes. It might look like this:

```
hello H EH L OW

world W ER L D
```

A dictionary can also contain alternative pronunciations. In that case you can designate them with a number in parentheses:

```
the TH IH

the(2) TH AH
```

There are various phonesets to represent phones, such as IPA or SAMPA. CMUSphinx does not yet require you to use any well-known phoneset, moreover, it prefers to use letter-only phone names without special symbols. This requirement simplifies some processing

algorithms, for example, you can create files with phone names as part of the filenames without any violating of the OS filename requirements.

A dictionary should contain all the words you are interested in, otherwise the recognizer will not be able to recognize them. However, it is not sufficient to have the words in the dictionary. The recognizer looks for a word in both the dictionary and the language model. Without the language model, a word will not be recognized, even if it is present in the dictionary.

There is no need to remove unused words from the dictionary unless you want to save memory, extra words in the dictionary do not affect accuracy.

Using existing dictionaries

There are a number of dictionaries which cover languages we support – CMUDict for US English, French, German, Russian, Dutch, Italian, Spanish and Mandarin. Other dictionaries might be found on the web. If a dictionary has a proper format you can use it.

If a dictionary does not cover all the words you are interested in, you can extend it with the *g2p*tool.

Using g2p-seq2seq to extend the dictionary

There are various tools to help you to extend an existing dictionary for new words or to build a new dictionary from scratch. Two of them are Phonetisaurus and Sequitur.

We recommend to use our latest tool g2p-seq2seq . It is based on neural networks implemented in the Tensorflow framework and provides a state-of-the-art accuracy of conversion.

An English model 2-layer LSTM with 512 hidden units is available for download on the CMUSphinx website. Unpack the model after downloading. It is trained on the CMU English dictionary. As the name says, this model works only for English. For other languages you first need to bootstrap a dictionary as described below and then use the G2P tool to extend it.

The easiest way to check how the G2P tool works is to run the interactive mode with the model from above:

```
g2p-seq2seq --interactive --model model_folder_path



> hello

HH EH L OW
```

To generate pronunciations for an English word list with a trained model, run:

```
g2p-seq2seq --decode your_wordlist --model model_folder_path
```

The wordlist is a text file with words, one word per line.

To train G2P you need a dictionary (a word and phone sequence per line in the standard form). Run the training with:

```
g2p-seq2seq --train train_dictionary.dic --model model_folder_path
```

For more information on the G2P tool have a look at the Readme of the G2P project.

Bootstrapping a dictionary for other languages

If you do not have a dictionary for your language there are usually several ways how you can create them.

Usually, dictionaries are bootstrapped with hand-written rules. You can find a list of phonemes for your language in the Wikipedia page about your language and write a simple Python script to map words to phonemes. The best dictionary could not be covered with rules though, most languages have quite irregular pronunciation which might not be very obvious for a newcomer even if it is conventionally thought that you speak what is written. This is due to coarticulation effects in human speech. However, for a basic dictionary, rules are sufficiently good enough.

You can crawl the Wiktionary to get a mapping for a significant amount of words covered there.

You can use TTS tools like from OpenMary written in Java or from Espeak written in C to create the phonetic dictionary for the languages they support.

Many languages which use hieroglyphs like Korean or Japanese have specialized software like Mecab to romanize their words. You can use Mecab to build a phonetic dictionary by converting words to the romanized form and then simply applying rules to turn them into phones.

It is enough to transcribe a few thousand most common words to bootstrap the dictionary.

Once your dictionary is bootstrapped you can extend it to hold a larger vocabulary with the g2p-seq2seq tool as described in the previous section.

Building a language model

The language model is an important component of the configuration which tells the decoder which sequences of words are possible to recognize.

There are several types of models: keyword lists, grammars and statistical language models and phonetic language models. They have different capabilities and performance properties. You can chose any decoding mode according to your needs and you can even switch between modes in runtime. See the Pocketsphinx tutorial for more details.

Keyword lists

Pocketsphinx supports a keyword spotting mode where you can specify a list of keywords to look for. The advantage of this mode is that you can specify a threshold for each keyword so that keywords can be detected in continuous speech. All other modes will try to detect the words from a grammar even if you used words which are not in the grammar. A typical keyword list looks like this:

```
oh mighty computer /1e-40/

hello world /1e-30/

other phrase /1e-20/
```

The threshold must be specified for every keyphrase. For shorter keyphrases you can use smaller thresholds like `1e-1`, for longer keyphrases the threshold must be bigger, up to `1e-50`. If your keyphrase is very long – larger than 10 syllables – it is recommended to split it and spot for parts separately. The threshold must be tuned to balance between false alarms and missed detections. The best way to do this is to use a prerecorded audio file. The common tuning process is the following:

1. Take a long recording with few occurrences of your keywords and some other sounds. You can take a movie sound or something else. The length of the audio should be approximately 1 hour.
2. Run a keyword spotting on that file with different thresholds for every keyword, use the following command:

```
3.  pocketsphinx_continuous -infile <your_file.wav> -keyphrase <your
    keyphrase> \

4.    -kws_threshold <your_threshold> -time yes
```

   The command will print many lines, some of them are keywords with detection times and confidences. You can also disable extra logs with the `-logfn your_file.log` option to avoid clutter.

5. From your keyword spotting results count how many false alarms and missed detections you've encountered.
6. Select the threshold with the smallest amount of false alarms and missed detections.

For the best accuracy it is better to have a keyphrase with 3-4 syllables. Too short phrases are easily confused.

Keyword lists are only supported by pocketsphinx, sphinx4 cannot handle them.

Using keyword lists with PocketSphinx

To use keyword list in the command line specify it with the `-kws` option. You can also use a `-keyphrase` option to specify a single keyphrase.

In Python you can either specify options in the configuration object or add a named search for a keyphrase:

```
decoder.set_kws('keyphrase', kws_file)

decoder.set_search('keyphrase')
```

In Android it looks similar:

```
recognizer.setKws('keyphrase', kwsFile);

recognizer.startListening('keyphrase')
```

Please note that `-kws` conflicts with the `-lm` and `-jsgf` options. You cannot specify both.

Grammars

A grammar describes a very simple type of the language for command and control. They are usually written by hand or generated automatically within the code. Grammars usually do not have probabilities for word sequences, but some elements might be weighed. They can be created with the Java Speech Grammar Format (JSGF) and usually have a file extension like *.gram* or *.jsgf*.

Grammars allow you to specify possible inputs very precisely, for example, that a certain word might be repeated only two or three times. However, this strictness might be harmful if your user accidentally skips the words which the grammar requires. In that case the whole recognition will fail. For that reason it is better to make grammars more flexible. Instead of phrases, just list the bag of words allowing arbitrary order. Avoid very complex grammars with many rules and cases. It just slows down the recognizer and you can use simple rules instead. In the past, grammars required a lot of effort to tune them, to assign variants properly and so on. The big VXML consulting industry was about that.

Building a grammar

Grammars are usually written manually in the Java Speech Grammar Format (JSGF):

```
#JSGF V1.0;



grammar hello;

public <greet> = (good morning | hello) ( bhiksha | evandro | rita | will );
```

For more information on JSGF see the full documentation on W3C.

Using your grammar with PocketSphinx

To use your grammar in the command line specify it with the `-jsgf` option.

In Python you can either specify options in the configuration object or add a named search for a grammar:

```
decoder.set_jsgf('grammar', jsgf_file)

decoder.set_search('grammar')
```

In Android this looks similiar:

```
recognizer.setJsgf('grammar', jsgfFile);

recognizer.startListening('grammar')
```

Please note that `-jsgf` conflicts with the `-kws` and `-jsgf` options. You cannot specify both.

Language models

Statistical language models describe more complex language. They contain probabilities of the words and word combinations. Those probabilities are estimated from sample data and automatically have some flexibility. Every combination from the vocabulary is possible, although the probability of each combination will vary. For example, if you create a statistical language model from a list of words it will still allow to decode word combinations even though this might not have been your intent.

Overall, statistical language models are recommended for free-form input where the user could say anything in a natural language. They require way less engineering effort than grammars. You just list the possible sentences. For example, you might list numbers like “twenty one” and “thirty three” and a statistical language model will allow “thirty one” with a certain probability as well.

In general, modern speech recognition interfaces tend to be more natural and avoid the command-and-control style of the previous generation. For that reason most interface designers prefer natural language recognition with a statistical language model instead of using old-fashioned VXML grammars.

On the topic of desiging VUI interfaces you might be interested in the following book: It’s Better to Be a Good Machine Than a Bad Person: Speech Recognition and Other Exotic User Interfaces at the Twilight of the Jetsonian Age by Bruce Balentine.

There are many ways to build statistical language models. When your data set is large, it makes sense to use the CMU language modeling toolkit. When a model is small, you can use a quick online web service. When you need specific options or you just want to use your favorite toolkit which builds ARPA models, you can use this as well.

A language model can be stored and loaded in three different formats: *text ARPA* format, *binary BIN*format and *binary DMP* format. The ARPA format

takes more space but it is possible to edit it. ARPA files have an `.lm` extension. Binary formats take significantly less space and load faster. Binary files have a `.lm.bin` extension. It is also possible to convert between these formats. The DMP format is obsolete and not recommended.

Building a statistical language model

Text preparation

First of all you need to prepare a large collection of clean texts. Expand abbreviations, convert numbers to words, clean non-word items. For example to clean Wikipedia XML dumps you can use special Python scripts like Wikiextractor. To clean HTML pages you can try BoilerPipe. It's a nice package specifically created to extract text from HTML.

For an example on how to create a language model from Wikipedia text, please see this blog post. Movie subtitles are also a good source for spoken language.

Once you have gone through the language modeling process, please submit your language model to the CMUSphinx project. We'll be happy to share it!

Language modeling for Mandarin and other similar languages, is largely the same as for English, with one additional consideration. The difference is that the input text must be word segmented. A segmentation tool and an associated word list is provided to accomplish this.

Training an ARPA model with SRILM

Training a model with the SRI Language Modeling Toolkit (SRILM) is easy. That's why we recommend it. Moreover, SRILM is the most advanced toolkit up to date. To train a model you can use the following command:

```
ngram-count -kndiscount -interpolate -text train-text.txt -lm your.lm
```

You can prune the model afterwards to reduce the size of the model:

```
ngram -lm your.lm -prune 1e-8 -write-lm your-pruned.lm
```

After training it is worth it to test the perplexity of the model on the test data:

```
ngram -lm your.lm -ppl test-text.txt
```

Training an ARPA model with CMUCLMTK

You need to download and install the language model toolkit for CMUSphinx (CMUCLMTK). See the download page for details.

The process for creating a language model is as follows:

1) Prepare a reference text that will be used to generate the language model. The language model toolkit expects its input to be in the form of normalized text files, with utterances delimited by `<s>`and `</s>` tags. A number of input filters are available for specific corpora such as Switchboard, ISL and NIST meetings, and HUB5 transcripts. The result should be the set of sentences that are bounded by the start and end markers of the sentence: `<s>` and `</s>`. Here's an example:

```
<s> generally cloudy today with scattered outbreaks of rain and drizzle

persistent and heavy at times </s>

<s> some dry intervals also with hazy sunshine especially in eastern parts in

the morning </s>

<s> highest temperatures nine to thirteen Celsius in a light or moderate mainly

east south east breeze </s>

<s> cloudy damp and misty today with spells of rain and drizzle in most places

much of this rain will be light and patchy but heavier rain may develop in the

west later </s>
```

More data will generate better language models. The `weather.txt` file from sphinx4 (used to generate the weather language model) contains nearly 100,000 sentences.

2) Generate the vocabulary file. This is a list of all the words in the file:

```
 text2wfreq < weather.txt | wfreq2vocab > weather.tmp.vocab
```

3) You may want to edit the vocabulary file to remove words (numbers, misspellings, names). If you find misspellings, it is a good idea to fix them in the input transcript.

4) If you want a closed vocabulary language model (a language model that has no provisions for unknown words), then you should remove sentences from your input transcript that contain words that are not in your vocabulary file.

5) Generate the ARPA format language model with the commands:

```
  text2idngram -vocab weather.vocab -idngram weather.idngram < weather.closed.txt

  idngram2lm -vocab_type 0 -idngram weather.idngram -vocab weather.vocab -arpa
weather.lm
```

6) Generate the CMU binary form (BIN):

```
 sphinx_lm_convert -i weather.lm -o weather.lm.bin
```

Building a simple language model using a web service

If your language is English and the text is small it's sometimes more convenient to use a web service to build it. Language models built in this way are quite functional for simple command and control tasks. First of all you need to create a corpus.

The "corpus" is just a list of sentences that you will use to train the language model. As an example, we will use a hypothetical voice control task for a mobile Internet device. We'd like to tell it things like "open browser", "new e-mail", "forward", "backward", "next window", "last window", "open music player", and so forth. So, we'll start by creating a file called `corpus.txt`:

```
open browser
```

```
new e-mail

forward

backward

next window

last window

open music player
```

Then go to the LMTool page. Simply click on the *"Browse..."* button, select the `corpus.txt` file you created, then click *"COMPILE KNOWLEDGE BASE"*.

You should see a page with some status messages, followed by a page entitled *"Sphinx knowledge base"*. This page will contain links entitled *"Dictionary"* and *"Language Model"*. Download these files and make a note of their names (they should consist of a 4-digit number followed by the extensions `.dic` and `.lm`). You can now test your newly created language model with PocketSphinx.

Using other language model toolkits

There are many toolkits that create an ARPA n-gram language model from text files.

Some toolkits you can try:

- IRSLM
- MITLM

If you are training a large vocabulary speech recognition system, the language model training is outlined in a separate page about large scale language models.

Once you have created an ARPA file you can convert the model to a binary format for faster loading.

Converting a model into the binary format

To quickly load large models you probably would like to convert them to a binary format that will save your decoder initialization time. That's not necessary with small models. Pocketsphinx and sphinx3 can handle both of them with the `-lm` option. Sphinx4 automatically detects the format by the extension of the lm file.

The ARPA format and BINARY format are mutually convertable. You can produce the other file with the `sphinx_lm_convert` command from sphinxbase:

```
sphinx_lm_convert -i model.lm -o model.lm.bin

sphinx_lm_convert -i model.lm.bin -ifmt bin -o model.lm -ofmt arpa
```

You can also convert old DMP models to a binary format this way.

In the next section we will deal with how to use, test, and improve the language model you created.

Using your language model with PocketSphinx

If you have installed PocketSphinx, you will have a program called `pocketsphinx_continuous`which can be run from the command line to recognize speech. Assuming it is installed under`/usr/local`, and your language model and dictionary are called `8521.dic` and `8521.lm` and placed in the current folder, try running the following command:

```
pocketsphinx_continuous -inmic yes -lm 8521.lm -dict 8521.dic
```

This will use your new language model, the dictionary and the default acoustic model. On Windows you also have to specify the acoustic model folder with the `-hmm` option:

```
bin/Release/pocketsphinx_continuous.exe -inmic yes -lm 8521.lm -dict 8521.dic -hmm
model/en-us/en-us
```

You will see a lot of diagnostic messages, followed by a pause, then the output *"READY…"*. Now you can try speaking some of the commands. It should be able to recognize them with full accuracy. If not, you may have problems with your microphone or sound card.

Using your language model with Sphinx4

In the Sphinx4 high-level API you need to specify the location of the language model in your Configuration:

```
configuration.setLanguageModelPath("file:8754.lm");
```

If the model is in the resources you can reference it with `"resource:URL"`:

```
configuration.setLanguageModelPath("resource:/com/example/8754.lm");
```

Also see the Sphinx4 tutorial for more details.

Adapting the default acoustic model

Caution!
This tutorial uses the 5 pre-alpha release.
It is not going to work for older versions.

This page describes how to do some simple acoustic model adaptation to improve speech recognition in your configuration. Please note that the adaptation doesn't necessary adapt for a particular speaker. It just improves the fit between the adaptation data and the model. For example you can adapt to your own voice to make dictation good, but you also can adapt to your particular recording environment, your audio transmission channel, your accent or accent of your users. You can use a model trained with clean broadcast data and telephone data to produce a telephone acoustic model by doing adaptation. Cross-language adaptation also make sense, you can for example adapt an English model to sounds of another language by creating a phoneset map and creating another language dictionary with an English phoneset.

The adaptation process takes transcribed data and improves the model you already have. It's more robust than training and could lead to good results even if your adaptation data is small. For example, it's enough to have 5 minutes of speech to significantly improve the dictation accuracy by adapting to the particular speaker.

The methods of adaptation are a bit different between PocketSphinx and Sphinx4 due to the different types of acoustic models used. For more technical information on that read the article about Acoustic Model Types.

Creating an adaptation corpus

The first thing you need to do is to create a corpus of adaptation data. The corpus will consist of

- a list of sentences
- a dictionary describing the pronunciation of all the words in that list of sentences
- a recording of you speaking each of those sentences

Required files

The actual set of sentences you use is somewhat arbitrary, but ideally it should have good coverage of the most frequently used words or phonemes in the set of sentences or the type of text you want to recognize. For example, if you want to recognize isolated commands, you need tor record them. If you want to recognize dictation, you need to record full sentences. For simple voice adaptation we have had good results simply by using sentences from the CMU ARCTIC text-to-speech databases. To that effect, here are the first 20 sentences from ARCTIC, a `.fileids` file, and a transcription file:

- arctic20.fileids
- arctic20.transcription

The sections below will refer to these files, so, if you want to follow along we recommend downloading these files now. You should also make sure that you have downloaded and compiled sphinxbase and sphinxtrain.

Recording your adaptation data

In case you are adapting to a single speaker you can record the adaptation data yourself. This is unfortunately a bit more complicated than it ought to be.
Basically, you need to record a single audio file for each sentence in the adaptation corpus, naming the files according to the names listed in `arctic20.transcription` and `arctic20.fileids`.

In addition, you need to make sure that you record at a *sampling rate of 16 kHz* (or 8 kHz if you adapt a telephone model) in *mono with a single channel*.

The simplest way would be to start a sound recorder like Audacity or Wavesurfer and read all sentences in one big audio file. Then you can cut the audio files on sentences in a text editor and make sure every sentence is saved in the corresponding file. The file structure should look like this:

```
arctic_0001.wav

arctic_0002.wav

.....

arctic_0019.wav

arctic20.fileids

arctic20.transcription
```

You should verify that these recordings sound okay. To do this, you can play them back with:

```
for i in *.wav; do play $i; done
```

If you already have a recording of the speaker, you can split it on sentences and create the `.fileids` and the `.transcription` files.

If you are adapting to a channel, accent or some other generic property of the audio, then you need to collect a little bit more recordings manually. For example, in a call center you can record and transcribe hundred calls and use them to improve the recognizer accuracy by means of adaptation.

Adapting the acoustic model

First we will copy the default acoustic model from PocketSphinx into the current directory in order to work on it. Assuming that you installed PocketSphinx under `/usr/local`, the acoustic model directory is `/usr/local/share/pocketsphinx/model/en-us/en-us`. Copy this directory to your working directory:

```
cp -a /usr/local/share/pocketsphinx/model/en-us/en-us .
```

Let’s also copy the dictionary and the langauge model for testing:

```
cp -a /usr/local/share/pocketsphinx/model/en-us/cmudict-en-us.dict .

cp -a /usr/local/share/pocketsphinx/model/en-us/en-us.lm.bin .
```

Generating acoustic feature files

In order to run the adaptation tools, you must generate a set of acoustic model feature files from these WAV audio recordings. This can be done with the `sphinx_fe` tool from SphinxBase. It is imperative that you make sure you are using the same acoustic parameters to extract these features as were used to train the standard acoustic model. Since PocketSphinx 0.4, these are stored in a file called `feat.params` in the acoustic model directory. You can simply add it to the command line for `sphinx_fe`, like this:

```
sphinx_fe -argfile en-us/feat.params \

        -samprate 16000 -c arctic20.fileids \

       -di . -do . -ei wav -eo mfc -mswav yes
```

You should now have the following files in your working directory:

```
en-us

arctic_0001.mfc

arctic_0001.wav

arctic_0002.mfc

arctic_0002.wav

arctic_0003.mfc

arctic_0003.wav

.....

arctic_0020.wav

arctic20.fileids

arctic20.transcription

cmudict-en-us.dict

en-us.lm.bin
```

Converting the sendump and mdef files

Some models like en-us are distributed in compressed version. Extra files that are required for adaptation are excluded to save space. For the en-us model from pocketsphinx you can download the full version suitable for adaptation:

cmusphinx-en-us-ptm-5.2.tar.gz

Make sure you are using the full model with the `mixture_weights` file present.

If the `mdef` file inside the model is converted to binary, you will also need to convert the `mdef` file from the acoustic model to the plain text format used by the SphinxTrain tools. To do this, use the `pocketsphinx_mdef_convert` program:

```
pocketsphinx_mdef_convert -text en-us/mdef en-us/mdef.txt
```

In the downloads the `mdef` is already in the text form.

Accumulating observation counts

The next step in the adaptation is to collect statistics from the adaptation data.
This is done using the `bw` program from SphinxTrain. You should be able to find the `bw` tool in a sphinxtrain installation in the folder `/usr/local/libexec/sphinxtrain` (or under another prefix on Linux) or in `bin\Release` (in the sphinxtrain directory on Windows). Copy it to the working directory along with the `map_adapt` and `mk_s2sendump` programs.

Now, to collect the statistics, run:

```
./bw \
 -hmmdir en-us \
 -moddeffn en-us/mdef.txt \
 -ts2cbfn .ptm. \
 -feat 1s_c_d_dd \
 -svspec 0-12/13-25/26-38 \
 -cmn current \
 -agc none \
 -dictfn cmudict-en-us.dict \
 -ctlfn arctic20.fileids \
 -lsnfn arctic20.transcription \
 -accumdir .
```

Make sure the arguments in the `bw` command match the parameters in the `feat.params` file inside the acoustic model folder. Please note that not all the parameters from `feat.param` are supported by `bw`. `bw` for example doesn't suppport `upperf` or other feature extraction parameters. You only need to use parameters which are accepted, other parameters from `feat.params` should be skipped.

For example, for a continuous model you don't need to include the `svspec` option. Instead, you need to use just `-ts2cbfn .cont.` For semi-continuous models use `-ts2cbfn .semi`. If the model has a `feature_transform` file like the en-us continuous model, you need to add the `-lda feature_transform` argument to `bw`, otherwise it will not work properly.

If you are missing the `noisedict` file, you also need an extra step. Copy the `fillerdict` file into the directory that you choose in the `hmmdir` parameter and renaming it to `noisedict`.

Creating a transformation with MLLR

MLLR transforms are supported by pocketsphinx and sphinx4. MLLR is a cheap adaptation method that is suitable when the amount of data is limited. It's a good idea to use MLLR for

online adaptation. MLLR works best for a continuous model. Its effect for semi-continuous models is very limited since semi-continuous models mostly rely on mixture weights. If you want the best accuracy you can combine MLLR adaptation with MAP adaptation below. On the other hand, because MAP requires a lot of adaptation data it is not really practical to use it for continuous models. For continuous models MLLR is more reasonable.

Next, we will generate an MLLR transformation which we will pass to the decoder to adapt the acoustic model at run-time. This is done with the `mllr_solve` program:

```
./mllr_solve \

    -meanfn en-us/means \

    -varfn en-us/variances \

    -outmllrfn mllr_matrix -accumdir .
```

This command will create an adaptation data file called `mllr_matrix`. Now, if you wish to decode with the adapted model, simply add `-mllr mllr_matrix` (or whatever the path to the mllr_matrix file you created is) to your pocketsphinx command line.

Updating the acoustic model files with MAP

MAP is a different adaptation method. In this case, unlike for MLLR, we don't create a generic transform but update each parameter in the model. We will now copy the acoustic model directory and overwrite the newly created directory with the adapted model files:

```
cp -a en-us en-us-adapt
```

To apply the adaptation, use the `map_adapt` program:

```
./map_adapt \

    -moddeffn en-us/mdef.txt \

    -ts2cbfn .ptm. \

    -meanfn en-us/means \

    -varfn en-us/variances \

    -mixwfn en-us/mixture_weights \

    -tmatfn en-us/transition_matrices \

    -accumdir . \

    -mapmeanfn en-us-adapt/means \

    -mapvarfn en-us-adapt/variances \

    -mapmixwfn en-us-adapt/mixture_weights \

    -maptmatfn en-us-adapt/transition_matrices
```

Recreating the adapted sendump file

If you want to save space for the model you can use a `sendump` file which is supported by PocketSphinx. For Sphinx4 you don't need that. To recreate the `sendump` file from the updated `mixture_weights` file run:

```
./mk_s2sendump \

    -pocketsphinx yes \

    -moddeffn en-us-adapt/mdef.txt \

    -mixwfn en-us-adapt/mixture_weights \

    -sendumpfn en-us-adapt/sendump
```

Congratulations! You now have an adapted acoustic model.

The `en-us-adapt/mixture_weights` and `en-us-adapt/mdef.txt` files are not used by the decoder, so, if you like, you can delete them to save some space.

Other acoustic models

For Sphinx4, the adaptation is the same as for PocketSphinx, except that Sphinx4 can not read the binary compressed `mdef` and `sendump` files, you need to leave the `mdef` and the `mixture weights` file.

Testing the adaptation

After you have done the adaptation, it's critical to test the adaptation quality. To do that you need to setup the database similar to the one used for adaptation. To test the adaptation you need to configure the decoding with the required paramters, in particular, you need to have a language model `<your.lm>`. For more details see the tutorial on Building a Language Model. The detailed process of testing the model is covered in another part of the tutorial.

You can try to run the decoder on the original acoustic model and on the new acoustic model to estimate the improvement.

Using the model

After adaptation, the acoustic model is located in the folder `en-us-adapt`. You need only that folder. The model should have the following files:

```
mdef

feat.params

mixture_weights

means

noisedict

transition_matrices
```

```
variances
```

depending on the type of the model you trained.

To use the model in PocketSphinx, simply put the model files to the resources of your application. Then point to it with the `-hmm` option:

```
pocketsphinx_continuous -hmm `<your_new_model_folder>` -lm `<your_lm>` -dict
`<your_dict>` -infile test.wav
```

or with the `-hmm` engine configuration option through the `cmd_ln_init` function. Alternatively, you can replace the old model files with the new ones.

To use the trained model in Sphinx4, you need to update the model location in the code.

Troubleshooting

If the adaptation didn't improve your results, first test the accuracy and make sure it's good.

I have no idea where to start looking for the problem…

1. Test whether the accuracy on the adaptation set improved
2. Accuracy improved on adaptation set ⇢ check if your adaptation set matches with your test set
3. Accuracy didn't improve on adaptation set ⇢ you made a mistake during the adaptation

…or how much improvement I might expect through adaptation

From few sentences you should get about 10% relative WER improvement.

I'm lost about…

…whether it needs more/better training data, whether I'm not doing the adaptation correctly, whether my language model is the problem here, or whether there is something intrinsically wrong with my configuration.

Most likely you just ignored some error messages that were printed to you. You obviosly need to provide more information and give access to your experiment files in order to get more definite advise.

What's next

We hope the adapted model gives you acceptable results. If not, try to improve your adaptation process by:

1. Adding more adaptation data
2. Adapting your language mode / using a better language model

Training an acoustic model for CMUSphinx

Introduction

The CMUSphinx project comes with several high-quality acoustic models. There are US English acoustic models for microphone and broadcast speech as well as a model for speech over a telephone. You can also use French or Chinese models trained on a huge amount of acoustic data. Those models were carefully optimized to achieve the best recognition performance and work well for almost all applications. We put years of experience into making them perfect. Most command-and-control applications and even some large vocabulary applications could just use default models directly.

Besides models, CMUSphinx provides some approaches for adaptation which should suffice for most cases when more accuracy is required. Adaptation is known to work well when you are using different recording environments (close-distance or far microphone or telephone channel), or when a slightly different accent (UK English or Indian English) or even another language is present. Adaptation, for example, works well if you need to quickly add support for some new language just by mapping a phoneset of an acoustic model to a target phoneset with the dictionary.

There are, however, applications where the current models won't work. Such examples are handwriting recognition or dictation support for another language. In these cases, you will need to train your own model and this tutorial demonstrates how to do the training for the CMUSphinx speech recognition engine. Before starting with the training make sure you are

familiar with the concepts, prepared the language model. Be sure that you indeed need to train the model and that you have the resources to do that.

When you need to train

You need to train an acoustic model if:

- You want to create an acoustic model for a new language or dialect
- OR you need a specialized model for a small vocabulary application
- *AND you have plenty of data to train on*:
    - 1 hour of recording for command and control for a single speaker
    - 5 hours of recordings of 200 speakers for command and control for many speakers
    - 10 hours of recordings for single speaker dictation
    - 50 hours of recordings of 200 speakers for many speakers dictation
- AND you have knowledge on the phonetic structure of the language
- AND you have time to train the model and optimize the parameters (~1 month)

When you don't need to train

You don't need to train an acoustic model if:

- You need to improve accuracy – do acoustic model adaptation instead
- You do not have enough data – do acoustic model adaptation instead
- You do not have enough time
- You do not have enough experience

Please note that the amounts of data listed here *are required* to train a model. If you have significantly less data than listed you can not expect to train a good model. For example, you *cannot* train a model with 1 minute of speech data.

Data preparation

The trainer learns the parameters for the models of the sound units using a set of sample speech signals. This is called a training database. A selection of already trained databases will also be provided to you.

The database contains information that is required to extract statistics from the speech in form of the acoustic model.

The trainer needs to be told which sound units you want it to learn the parameters of, and at least the sequence in which they occur in every speech signal in your training database. This information is provided to the trainer through a file called the *transcript file*. It contains the sequence of words and non-speech sounds in the exact same order as they occurred in a speech signal, followed by a tag which can be used to associate this sequence with the corresponding speech signal.

Thus, in addition to the speech signals and transcription file, the trainer needs to access two dictionaries: one in which legitimate words in the language are mapped to sequences of sound units (or sub-word units), and a another one in which non-speech sounds are mapped to corresponding non-speech or speech-like sound units. We will refer to the former as the language *phonetic dictionary*. The latter is called the *filler dictionary*. The trainer then looks

into the dictionaries, to derive the sequence of sound units that are associated with each signal and transcription.

After training, it's mandatory to run the decoder to check the training results. The Decoder takes a model, tests part of the database and reference transcriptions and estimates the quality (WER) of the model. During the testing stage we use the *language model* with the description of the possible order of words in the language.

To setup a training, you first need to design a training database or download an existing one. For example, you can purchase a database from LDC. You'll have to convert it into a proper format.

A database should be a good representation of what speech you are going to recognize. For example if you are going to recognize telephone speech it is preferred to use telephone recordings. If you want to work with mobile speech, you should better find mobile recordings. Speech is significantly different across various recording channels. Broadcast news is different from a phone call. Speech decoded from mp3 is significantly different from a microphone recording. However, if you do not have enough speech recorded in the required conditions you should definitely use any other speech you have. For example you can use broadcast recordings. Sometimes it make sense to stream through the telephone codec to equalize the audio. It's often possible to add noise to training data too.

The database should have recording of enough speakers, a variety of recording conditions, enough acoustic variations and all possible linguistic sentences. As mentioned before, The size of the database depends on the complexity of the task you want to handle.

A database should have the two parts mentioned above: a training part and a test part. Usually the test part is about 1/10th of the total data size, but we don't recommend you to have more than 4 hours of recordings as test data.

Good approaches to obtain a database for a new language are:

- Manually segmenting audio recordings with existing transcription (podcasts, news, etc.)
- Recording your friends and family and colleagues
- Setting up an automated collection on Voxforge

You have to design database prompts and post-process the results to ensure that the audio actually corresponds to the prompts. The file structure for the database is the following:

```
├─ etc
│   ├─ your_db.dic                  (Phonetic dictionary)
│   ├─ your_db.phone                (Phoneset file)
│   ├─ your_db.lm.DMP               (Language model)
│   ├─ your_db.filler               (List of fillers)
│   ├─ your_db_train.fileids        (List of files for training)
│   ├─ your_db_train.transcription (Transcription for training)
```

```
|   ├─ your_db_test.fileids        (List of files for testing)
|   └─ your_db_test.transcription  (Transcription for testing)
└─ wav
    ├─ speaker_1
    |    └─ file_1.wav             (Recording of speech utterance)
    └─ speaker_2
         └─ file_2.wav
```

Let's go through the files and describe their format and the way to prepare them:

***.fileids:** The `your_db_train.fileids` and `your_db_test.fileids` files are text files which list the names of the recordings (utterance ids) one by one, for example:

```
speaker_1/file_1

speaker_2/file_2
```

A **.fileids* file contains the path in a file-system relative to the *wav* directory. Note that a **.fileids*file should not include audio file extensions in its content, but rather just the names.

***.transcription:** The `your_db_train.transcription` and `your_db_test.transcription` files are text files listing the transcription for each audio file:

```
<s> hello world </s> (file_1)

<s> foo bar </s> (file_2)
```

It's important that each line starts with `<s>` and ends with `</s>` followed by an id in parentheses. Also note that the parentheses contains only the file, without the *speaker_n* directory. It's critical to have an exact match between the **.fileids* file and the **.transcription* file. The number of lines in both should be identical. The last part of the file id `(speaker1/file_1)` and the utterance id `file_1`must be the same on each line.

Below is an example of an *incorrect *.fileids* file for the above transcription file. If you follow it, you will get an error as discussed here:

```
speaker_2/file_2

speaker_1/file_1

// Bad! Do not create *.fileids files like this!
```

Speech recordings (*.wav files): Your audio recordings should contain training audio which should match the audio you want to recognize in the end. In case of a mismatch you could experience a sometimes even significant drop of the accuracy. This means if you want to recognize continuous speech, your training database should record continuous speech. For continuous speech the optimal length for audio recordings is between 5 seconds and 30 seconds. Very long recordings make training much harder. If you are going to recognize short isolated commands, your training database should also contain files with short isolated commands. It is better to design the database to recognize continuous speech from the

beginning though and not spend your time on commands. In the end you speak continuously anyway. The Amount of silence in the beginning and in the end of the utterance should not exceed 200 ms.

Recording files must be in *MS WAV* format with a specific sample rate – 16 kHz, 16 bit, mono for desktop application, 8kHz, 16bit, mono for telephone applications. It's *critical* that the audio files have a specific format. Sphinxtrain does support some variety of sample rates but by default it is configured to train from *16khz 16bit mono* files in MS WAV format.

So, please make sure that your recordings have a *samplig rate of 16 kHz* (or 8 kHz if you train a telephone model) in *mono* with a *single channel*!

If you train from an 8 kHz model you need to make sure you configured the feature extraction properly. Please note that you *cannot upsample* your audio, that means you can not train 16 kHz model with 8 kHz data.

A mismatch of the audio format is the most common training problem – make sure you eliminated this source of problems.

Phonetic Dictionary (your_db.dict): should have one line per word with the word following the phonetic transcription:

```
HELLO HH AH L OW

WORLD W AO R L D
```

If you need to find a phonetic dictionary, have a look on Wikipedia or read a book on phonetics. If you are using an existing phonetic dictionary do not use case-sensitive variants like "e" and "E". Instead, all your phones must be different even in the case-insensitive variation. Sphinxtrain doesn't support some special characters like "*" or "/" and supports most of others like "+", "-" or ":". However, to be safe we recommend you to use alphanumeric-only phone-set.

Replace special characters in the phone-set, like colons, dashes or tildes, with something alphanumeric. For example, replace "a~" with "aa" to make it alphanumeric only. Nowadays, even cell phones have gigabytes of memory on board. There is no sense in trying to save space with cryptic special characters.

There is one very important thing here. For a large vocabulary database, the phonetic representation is more or less known; it's simple phones described in any book. If you don't have a phonetic book, you can just use the word's spelling and it will also give you very good results:

```
ONE O N E

TWO T W O
```

For a small vocabulary CMUSphinx is different from other toolkits. It's often recommended to train word-based models for a small vocabulary databases like digits. Yet, this only makes sense if your HMMs could have variable length.

CMUSphinx does not support word models. Instead, you need to use a word-dependent phone dictionary:

```
ONE W_ONE AH_ONE N_ONE

TWO T_TWO UH_TWO

NINE N_NINE AY_NINE N_END_NINE
```

This is actually *equivalent* to word-based models and some times even gives better accuracy. *Do not use word-based models with CMUSphinx!*

Phoneset file (your_db.phone): should have one phone per line. The number of phones should match the phones used in the dictionary plus the special SIL phone for silence:

```
AH

AX

DH

IX
```

Language model file (your_db.lm.DMP): should be in ARPA format or in DMP format. Find our more about language models in the Building a language model chapter.

Filler dictionary (your_db.filler): contains filler phones (not-covered by language model non-linguistic sounds like breath, “hmm” or laugh). It can contain just silences:

```
<s> SIL

</s> SIL

<sil> SIL
```

It can also contain filler phones if they are present in the database transcriptions:

```
+um+ ++um++

+noise+ ++noise++
```

The sample database for training is available at an4 database NIST’s Sphere audio (.sph) format, You can use this database in the following sections. If you want to play with a large example, download the TED-LIUM English acoustic database. It contains about 200 hours of audio recordings at present.

Compilation of the required packages

The following packages are required for training:

- sphinxbase-5prealpha
- sphinxtrain-5prealpha
- pocketsphinx-5prealpha

The following external packages are also required:

- perl, for example ActivePerl on Windows
- python, for example ActivePython on Windows

In addition, if you download the packages with a `.gz` suffix, you will need `gunzip` or an equivalent tool to unpack them.

Install the perl and python packages somewhere in your executable path, if they are not already there.

We recommend that you train on Linux: this way you'll be able to use all the features of sphinxtrain. You can also use a Windows system for training, in that case we recommend to use ActivePerl.

For further download instructions, see the download page.

Basically you need to put everything into a single root folder, unzip and untar them, and run `configure` and `make` and `make install` in each package folder. Put the database folder into this root folder as well. By the time you finish this, you will have a tutorial directory with the following contents:

```
└─ tutorial
   ├─ an4
   ├─ an4_sphere.tar.gz
   ├─ sphinxtrain
   ├─ sphinxtrain-5prealpha.tar.gz
   ├─ pocketsphinx
   ├─ pocketsphinx-5prealpha.tar.gz
   ├─ sphinxbase
   └─ sphinxbase-5prealpha.tar.gz
```

You will need to install the software as an administrator `root`. After you installed the software you may need to update the system configuration so that the system will be able to find the dynamic libraries, e.g.:

```
export PATH=/usr/local/bin:$PATH

export LD_LIBRARY_PATH=/usr/local/lib

export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
```

If you don't want to install into your system path, you may install the packages in your home folder. In that case you can append the following option to the `autogen.sh` script or to the `configure` script:

```
--prefix=/home/user/local
```

Obviously, the folder can be an arbitrary folder, just remember to update the environment configuration after modifying its name. If your binaries fail to load dynamic libraries with an error message like `failed to open libsphinx.so.0 no such file or directory`, it means that you didn't configure the environment properly.

Setting up the training scripts

To start the training, change to the database folder and run the following commands:

On Linux:

```
sphinxtrain -t an4 setup
```

On Windows:

```
python ../sphinxtrain/scripts/sphinxtrain -t an4 setup
```

Do not forget to replace *an4* with your task name.

This will copy all the required configuration files into the *etc/* subfolder of your database folder and will prepare the database for training. The directory structure after the setup will look like this:

```
├─ etc
└─ wav
```

In the process of the training other data folders will be created, so that your database directory should look like this:

```
├─ etc
├─ feat
├─ logdir
├─ model_parameters
├─ model_architecture
├─ result
└─ wav
```

After this basic setup, we need to edit the configuration files in the *etc/* folder. There are many variables but to get started we need to change only a few. First of all, find the file `etc/sphinx_train.cfg`.

Setting up the format of database audio

In `etc/sphinx_train.cfg` you should see the following configurations:

```
$CFG_WAVFILES_DIR = "$CFG_BASE_DIR/wav";
$CFG_WAVFILE_EXTENSION = 'sph';
```

```
$CFG_WAVFILE_TYPE = 'nist'; # one of nist, mswav, raw
```

If you recorded audio in WAV format, change `sph` to `wav` here and `nist` to `mswav`:

```
$CFG_WAVFILES_DIR = "$CFG_BASE_DIR/wav";

$CFG_WAVFILE_EXTENSION = 'wav';

$CFG_WAVFILE_TYPE = 'mswav'; # one of nist, mswav, raw
```

Configuring file paths

Search for the following lines in your `etc/sphinx_train.cfg` file:

```
# Variables used in main training of models

$CFG_DICTIONARY     = "$CFG_LIST_DIR/$CFG_DB_NAME.dic";

$CFG_RAWPHONEFILE   = "$CFG_LIST_DIR/$CFG_DB_NAME.phone";

$CFG_FILLERDICT     = "$CFG_LIST_DIR/$CFG_DB_NAME.filler";

$CFG_LISTOFFILES    = "$CFG_LIST_DIR/${CFG_DB_NAME}_train.fileids";

$CFG_TRANSCRIPTFILE = "$CFG_LIST_DIR/${CFG_DB_NAME}_train.transcription"
```

These values would be already set if you set up the file structure like described earlier, but make sure that your files are really named this way.

The `$CFG_LIST_DIR` variable is the `/etc` directory in your project. The `$CFG_DB_NAME` variable is the name of your project itself.

Configuring model type and model parameters

To select the acoustic model type see the Acoustic Model Types article.

```
$CFG_HMM_TYPE = '.cont.'; # Sphinx4, Pocketsphinx

#$CFG_HMM_TYPE  = '.semi.'; # PocketSphinx only

#$CFG_HMM_TYPE  = '.ptm.'; # Sphinx4, Pocketsphinx, faster model
```

Just uncomment what you need. For resource-efficient applications use semi-continuous models, for best accuracy use continuous models. By default we use PTM models which provide a nice balance between accuracy and speed.

```
$CFG_FINAL_NUM_DENSITIES = 8;
```

If you are training continuous models for a large vocabulary and have more than 100 hours of data, put 32 here. It can be any power of 2: 4, 8, 16, 32, 64.

If you are training semi-continuous or PTM model, use 256 gaussians.

```
# Number of tied states (senones) to create in decision-tree clustering

$CFG_N_TIED_STATES = 1000;
```

This value is the number of senones to train in a model. The more senones a model has, the more precisely it discriminates the sounds. On the other hand, if you have too many senones, the model will not be generic enough to recognize yet unseen speech. That means that the WER will be higher on unseen data. That's why it is important to *not overtrain the models*. In case there are too many unseen senones, the warnings will be generated in the norm log on stage 50 below:

```
ERROR: "gauden.c", line 1700: Variance (mgau= 948, feat= 0, density=3,

component=38) is less then 0. Most probably the number of senones is too

high for such a small training database. Use smaller $CFG_N_TIED_STATES.
```

The approximate number of senones and the number of densities for a continuous model is provided in the table below:

Vocabulary	Audio in database / hours	Senones	Densities	Exam
20	5	200	8	Tidigits Digits Recognitio
100	20	2000	8	RM1 Command and Con
5000	30	4000	16	WSJ1 5k Small Dictation
20000	80	4000	32	WSJ1 20k Big Dictation
60000	200	6000	16	HUB4 Broadcast News
60000	2000	12000	64	Fisher Rich Telephone Tr

For semi-continuous and PTM models use a fixed number of 256 densities.

Of course you also need to understand that only senones that are present in transcription can be trained. It means that if your transcription isn't generic enough, e.g. if it's the same single word spoken by 10.000 speakers 10.000 times you still have just a few senones no matter how many hours of speech you recorded. In that case you just need a few senones in the model, not thousands of them.

Though it might seem that diversity could improve the model that's not the case. Diverse speech requires some artificial speech prompts and that decreases the speech naturalness. Artificial models don't help in real life decoding. In order to build the best database you need to try to reproduce the real environment as much as possible. It's even better to collect more speech to try to optimize the database size.

It's important to remember, that *optimal numbers depend on your database*. To train a model properly, you need to experiment with different values and try to select the ones which result in the best WER for a development set. You can experiment with the number of senones and the number of Gaussian mixtures at least. Sometimes it's also worth to experiment with the phoneset or the number of estimation iterations.

Configuring sound feature parameters

The default for sound files used in Sphinx is a rate of 16 thousand samples per second (16 KHz). If this is the case, the *etc/feat.params* file will be automatically generated with the recommended values.

If you are using sound files with a sampling rate of 8 kHz (telephone audio), you need to change some values in *etc/sphinx_train.cfg*. The lower sampling rate also means a change in the sound frequency ranges and the number of filters that are used to recognize speech. Recommended values are:

```
# Feature extraction parameters

$CFG_WAVFILE_SRATE = 8000.0;

$CFG_NUM_FILT = 31; # For wideband speech it's 40, for telephone 8khz reasonable
value is 31

$CFG_LO_FILT = 200; # For telephone 8kHz speech value is 200

$CFG_HI_FILT = 3500; # For telephone 8kHz speech value is 3500
```

Configuring parallel jobs to speedup the training

If you are on a multicore machine or in a PBS cluster you can run the training in parallel. The following options should do the trick:

```
# Queue::POSIX for multiple CPUs on a local machine

# Queue::PBS to use a PBS/TORQUE queue

$CFG_QUEUE_TYPE = "Queue";
```

Change the type to "Queue::POSIX" to run on multicore. Then change the number of parallel processes to run:

```
# How many parts to run Forward-Backward estimation in

$CFG_NPART = 1;

$DEC_CFG_NPART = 1; #  Define how many pieces to split decode in
```

If you are running on an 8-core machine start around 10 parts to fully load the CPU during training.

Configuring decoding parameters

Open `etc/sphinx_train.cfg` and make sure the following configurations are set:

```
$DEC_CFG_DICTIONARY     = "$CFG_BASE_DIR/etc/$CFG_DB_NAME.dic";
```

```
$DEC_CFG_FILLERDICT     = "$CFG_BASE_DIR/etc/$CFG_DB_NAME.filler";

$DEC_CFG_LISTOFFILES    = "$CFG_BASE_DIR/etc/${CFG_DB_NAME}_test.fileids";

$DEC_CFG_TRANSCRIPTFILE = "$CFG_BASE_DIR/etc/${CFG_DB_NAME}_test.transcription";

$DEC_CFG_RESULT_DIR     = "$CFG_BASE_DIR/result";


# These variables are used by the decoder and have to be defined by the user.

# They may affect the decoder output.


$DEC_CFG_LANGUAGEMODEL  = "$CFG_BASE_DIR/etc/${CFG_DB_NAME}.lm.DMP";
```

If you are training with *an4* please make sure that you changed *${CFG_DB_NAME}.lm.DMP* to *an4.ug.lm.DMP* since the name of the language model is different in an4 database:

```
$DEC_CFG_LANGUAGEMODEL  = "$CFG_BASE_DIR/etc/an4.ug.lm.DMP";
```

If everything is OK, you can proceed to training.

Training

First of all, go to the database directory:

```
cd an4
```

To train, just run the following commands:

On Linux:

```
sphinxtrain run
```

On Windows:

```
python ../sphinxtrain/scripts/sphinxtrain run
```

and it will go through all the required stages. It will take a few minutes to train. On large databases, training could take up to a month.

The most important stage is the first one which checks that everything is configured correctly and your input data is consistent.

Do not ignore the errors reported on the first 00.verify_all step!

The typical output during decoding will look like:

```
Baum Welch starting for 2 Gaussian(s), iteration: 3 (1 of 1)

0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
```

```
Normalization for iteration: 3

Current Overall Likelihood Per Frame = 30.6558644286942

Convergence Ratio = 0.633864444461992

Baum Welch starting for 2 Gaussian(s), iteration: 4 (1 of 1)

0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%

Normalization for iteration: 4
```

These scripts process all required steps to train the model. After they finished, the training is complete.

Training internals

This section describes in detail what happens during the training.

In the scripts directory (`./scripts_pl`), there are several directories numbered sequentially from *00* through *99*. Each directory either has a directory named `slave*.pl` or it has a single file with the extension `.pl`. The script sequentially goes through the directories and executes either the`slave*.pl` or the single `.pl` file, as below.

```
perl scripts_pl/000.comp_feat/slave_feat.pl

perl scripts_pl/00.verify/verify_all.pl

perl scripts_pl/10.vector_quantize/slave.VQ.pl

perl scripts_pl/20.ci_hmm/slave_convg.pl

perl scripts_pl/30.cd_hmm_untied/slave_convg.pl

perl scripts_pl/40.buildtrees/slave.treebuilder.pl

perl scripts_pl/45.prunetree/slave-state-tying.pl

perl scripts_pl/50.cd_hmm_tied/slave_convg.pl

perl scripts_pl/90.deleted_interpolation/deleted_interpolation.pl
```

These scripts launch jobs on your machine, and the jobs will take a few minutes each to run through.

Before you run any script, note the directory contents of your current directory. After you run each `slave.pl` look at the the contents again. Several new directories will have been created. These directories contain files which are being generated in the course of your training. At this point you don’t need to know about the contents of these directories, though some of the directory names may be self-explanatory and you may explore them if you are curious.

One of the files that appears in your current directory is an *.html* file, named *an4.html*, depending on which database you are using. This file will contain a status report of the already executed jobs. Verify that the job you launched completed successfully. Only then launch the next *slave.pl* in the specified order. Repeat this process until you have run the *slave.pl* in all directories.

Note that in the process of going through the scripts from 00 to 90, you will have generated several sets of acoustic models, each of which could be used for recognition. Notice also that some of the steps are required only for the creation of semi-continuous models. If you execute these steps while creating continuous models, the scripts will benignly do nothing.

In the stage `000.comp_feat` the feature files are extracted. The system does not directly work with acoustic signals. The signals are first transformed into a sequence of feature vectors, which are used in place of the actual acoustic signals.

This script `slave_feat.pl` will compute a sequence of 13-dimensional vectors (feature vectors) for each training utterance consisting of the Mel Frequency Cepstral Coefficients (MFCCs). Note that the list of wave files contains a list with the full paths to the audio files. Since the data is all located in the same directory as your working directory, the paths are relative, not absolute. If the location of data is different, you may have to change this, as well as the *an4_test.fileids* file. The MFCCs will be placed automatically in a directory called `feat`. Note that the type of features vectors you compute from the speech signals for training and recognition, outside of this tutorial, is not restricted to MFCCs. You could use any reasonable parameterization technique instead, and compute features other than MFCCs. CMUSphinx can use features of any type or dimensionality. The format of the features is described on the MFC Format page.

Once the jobs launched from `20.ci_hmm` have run to completion, you will have trained the *Context-Independent* (CI) models for the sub-word units in your dictionary.

When the jobs launched from the `30.cd_hmm_untied` directory run to completion, you will have trained the models for *Context-Dependent* sub-word units (triphones) with untied states. These are called *CD-untied models* and are necessary for building decision trees in order to tie states.

The jobs in `40.buildtrees` will build decision trees for each state of each sub-word unit.

The jobs in `45.prunetree` will prune the decision trees and tie the states.

Following this, the jobs in `50.cd-hmm_tied` will train the final models for the triphones in your training corpus. These are called *CD-tied models*. The CD-tied models are trained in many stages. We begin with 1 Gaussian per state HMMs, followed by training 2 Gaussian per state HMMs and so on till the desired number of Gaussians per State have been trained. The jobs in `50.cd-hmm_tied`will automatically train all these intermediate CD-tied models.

At the end of any stage you may use the models for recognition. Remember that you may decode even while the training is in progress, provided you are certain that you have crossed the stage which generates the models you want to decode with.

Transformation matrix training (advanced)

Some additional scripts will be launched if you choose to run them. These additional training steps can be costly in computation, but improve the recognition rate.

Transform matrices might help the training and recognition process in some circumstances.

The following steps will run if you specify

```
$CFG_LDA_MLLT = 'yes';
```

in the file `sphinx_train.cfg`. If you specify `'no'` (which is the default), the steps will do nothing. The Step `01.lda_train` will estimate a LDA matrix and the step `02.mllt_train` will estimate a MLLT matrix.

The Perl scripts, in turn, set up and run Python modules. The end product for these steps is a `feature_transform` file, in your `model_parameters` directory. For details see the page for Training with LDA and MLLT.

MMIE training (advanced)

Finally, one more step will run if you specify MMIE training by setting:

```
$CFG_MMIE =  "yes";
```

The default value is `"no"`. This will run steps `60.lattice_generation`, `61.lattice_pruning`, `62.lattice_conversion` and `65.mmie_train`. For details see the page about MMIE Training in SphinxTrain.

Testing

It's *critical* to test the quality of the trained database in order to select the best parameters, understand how your application performs and optimize the performance. To do that, a test decoding step is needed. The decoding is now a last stage of the training process.

You can restart decoding with the following command:

```
sphinxtrain -s decode run
```

This command will start a decoding process using the acoustic model you trained and the language model you configured in the `etc/sphinx_train.cfg` file.

```
MODULE: DECODE Decoding using models previously trained

Decoding 130 segments starting at 0 (part 1 of 1)

0%
```

When the recognition job is complete, the script computes the recognition Word Error Rate (WER) and the Sentence Error Rate (SER). The lower those rates the better is your recognition. For a typical 10-hours task the WER should be around 10%. For a large task, it could be like 30%.

On an4 data you should get something like:

```
SENTENCE ERROR: 70.8% (92/130)   WORD ERROR RATE: 30.3% (233/773)
```

You can find exact details of the decoding, like the alignment with a reference transcription, speed and the result for each file, in the `result` folder which will be created after decoding. Let's have a look into the file `an4.align`:

```
p   I   T     t   s   b   u   r   g   H     (MMXG-CEN5-MMXG-B)
```

```
p   R   EIGHTY t   s   b   u   r   g   EIGHT  (MMXG-CEN5-MMXG-B)

Words: 10 Correct: 7 Errors: 3 Percent correct = 70.00% Error = 30.00% Accuracy =
70.00%

Insertions: 0 Deletions: 0 Substitutions: 3

october twenty four nineteen seventy  (MMXG-CEN8-MMXG-B)

october twenty four nineteen seventy  (MMXG-CEN8-MMXG-B)

Words: 5 Correct: 5 Errors: 0 Percent correct = 100.00% Error = 0.00% Accuracy =
100.00%

Insertions: 0 Deletions: 0 Substitutions: 0

TOTAL Words: 773 Correct: 587 Errors: 234

TOTAL Percent correct = 75.94% Error = 30.27% Accuracy = 69.73%

TOTAL Insertions: 48 Deletions: 15 Substitutions: 171
```

For a description of the WER see our Basic concepts of speech chapter.

Using the model

After training, the acoustic model is located in

```
model_parameters/`<your_db_name>`.cd_cont_`<number_of senones>`
```

or in

```
model_parameters/`<your_db_name>`.cd_semi_`<number_of senones>`
```

You need only that folder. The model should have the following files:

```
mdef

feat.params

mixture_weights

means

noisedict

transition_matrices

variances
```

depending on the type of the model you trained. To use the model in PocketSphinx, simply point to it with the `-hmm` option:

```
pocketsphinx_continuous -hmm `<your_new_model_folder>` -lm `<your_lm>` -dict
`<your_dict>`.
```

To use the trained model in Sphinx4, you need to specify the path in the `Configuration` object:

```
configuration.setAcousticModelPath("file:model_parameters/db.cd_cont_200");
```

If the model is in the resources you can reference it with `resource:URL`:

```
configuration.setAcousticModelPath("resource:/com/example/db.cd_cont_200");
```

See the Sphinx4 tutorial for details.

Troubleshooting

Troubleshooting is not rocket science. For all issues you may blame yourself. You are most likely the reason of failure. *Carefully* read the messages in the `logdir` folder that contains a detailed log for each performed action. In addition, messages are copied to the `your_project_name.html` file, which you can open and read in a browser.

There are many well-working, proven methods to solve issues. For example, try to reduce the training set to see in which half the problem appears.

Here are some common problems:

- **WARNING: this phone (something) appears in the dictionary (dictionary file name), but not in the phone list (phone file name).**

  Your dictionary either contains a mistake, or you have left out a phone symbol in the phone file. You may have to delete any comment lines from your dictionary file.

- **WARNING: This word (word) has duplicate entries in (dictionary file name). Check for duplicates.**

  You may have to sort your dictionary file lines to find them. Perhaps a word is defined in both upper and lower case forms.

- **WARNING: This word: word was in the transcript file, but is not in the dictionary (transcript line) Do cases match?**

  Make sure that all the words in the transcript are in the dictionary, and that they have matching cases when they appear. Also, words in the transcript may be misspelled, run together or be a number or symbol that is not in the dictionary. If the dictionary file is not perfectly sorted, some entries might be skipped while looking for words. If you hand-edited the dictionary file, be sure that each entry is in the proper format.

  You may have specified phones in the phone list that are not represented in the words in the transcript. The trainer expects to find examples of each phone at least once.

- **WARNING: CTL file, audio file name.mfc, does not exist, or is empty.**

  The *.mfc* files are the feature files converted from the input audio files in stage `000.comp_feats`. Did you skip this step? Did you add new audio files without converting them? The training process expects a feature file to be there, but it isn't.

- **Very low recognition accuracy**

This might happen if there is a mismatch in the audio files and the parameters of training, or between the training and the testing.

- **ERROR: “backward.c”, line 430: Failed to align audio to transcript: final state of the search is not reached.**

  Sometimes audio in your database doesn’t match the transcription properly. For example the transcription file has the line “Hello world” but in audio actually “Hello hello world” is pronounced. The training process usually detects that and emits this message in the logs. If there are too many of such errors it most likely means you misconfigured something, e.g. you had a mismatch between audio and the text caused by transcription reordering. If there are few errors, you can ignore them. You might want to edit the transcription file to put the exact word which was pronounced. In the case above you need to edit the transcription file and put “Hello hello world” on the corresponding line. You might want to filter such prompts because they affect the quality of the acoustic model. In that case you need to enable the forced alignment stage during training. To do that edit the following line in `sphinx_train.cfg`:

  ```
  $CFG_FORCEDALIGN = 'yes';
  ```

  and run the training again. It will execute stages 10 and 11 and will filter your database.

- **Can’t open */*-1-1.match word_align.pl failed with error code 65280**

  This error occurs because the decoder did not run properly after the training. First check if the correct executable is present in your `PATH`. The executable shouldbe `pocketsphinx_batch` if the decoding script being used is `psdecode.pl` as set by the `$DEC_CFG_SCRIPT` variable in`sphinx_train.cfg`. On Linux run:

  ```
  which pocketsphinx_batch
  ```

  and see if it is located as expected. If it is not, you need to set the `PATH` variable properly. Similarly on Windows, run:

  ```
  where pocketsphinx_batch
  ```

  If the path to the decoding executable is set properly, read the log files in `logdir/decode/` to find out other reasons for the error.

- **To ask for help**

  If you want to ask for help about training, try to provide the training folder or at least the logdir. Pack the files into an archive and upload it to a public file sharing resource. Then post the link to the resource. Remember: the more information you provide the faster you will solve the problem.

Tuning speech recognition accuracy

Speech recognition accuracy is not always great.

First, it is important to understand whether your accuracy is just lower than expected or whether it is very low in general. If the accuracy is very low in general, you most likely misconfigured the decoder. If it is lower than expected, you can apply various ways to improve it.

The first thing you should do is to collect a database of test samples and measure the recognition accuracy. You need to dump utterances into wav files, write a reference text and use the decoder to decode it. Then calculate the Word Error Rate (WER) using the `word_align.pl` tool from Sphinxtrain. The size of the test database depends on the accuracy but usually it's sufficient to have 30 minutes of transcribed audio to test recognizer accuracy reliably.

Only if you have a test database you can proceed with optimizing the recognition accuracy.

Reasons for bad accuracy

The top reasons for a bad accuracy are:

- The mismatch of the sample rate and the number of channels of the incoming audio or the mismatch of the incoming audio bandwidth. It must be a 16 kHz (or 8 kHz, depending on the training data), 16bit Mono (= single channel) Little-Endian file. You need to fix the sample rate of the source with resampling (only if its rate is higher than that of the training data). You should not upsample a file and decode it with acoustic models trained on audio with a higher sampling rate. The audio file format (sampling rate, number of channels) can be verified using the command
  - `sox --i /path/to/audio/file`

  Find more information here: What is sample rate?

- The mismatch of the acoustic model. To verify this hypothesis you need to construct a language model from the test database text. Such a language model will be very good and must give you a high accuracy. If accuracy is still low, you need to work more on the acoustic model. You can use acoustic model adaptation to improve accuracy.
- The mismatch of the langauge model. You can create your own language model to match the vocabulary you are trying to decode.
- The mismatch in the dictionary and the pronuncation of the words. In that case some work must be done in the phonetic dictionary.

Test database setup

To test the recognition you need to configure the decoding with the required paramters, in particular, you need to have a language model `<your.lm>`. For more details see the Building a language model page.

Create a *fileids* file `test.fileids`:

```
test1

test2
```

Create a transcription file `test.transcription`:

```
some text (test1)

some text (test2)
```

Put the audio files in the *wav* folder. Make sure those files have a proper format and sample rate.

```
└─ wav
   ├─ test1.wav
   └─ test2.wav
```

Running the test

Now, let's run the decoder:

```
pocketsphinx_batch \

 -adcin yes \

 -cepdir wav \

 -cepext .wav \

 -ctl test.fileids \

 -lm `<your.lm>` \    # for example en-us.lm.bin from pocketsphinx

 -dict `<your.dic>` \ # for example cmudict-en-us.dict from pocketsphinx

 -hmm `<your_hmm>` \  # for example en-us

 -hyp test.hyp


word_align.pl test.transcription test.hyp
```

The `word_align.pl` script is a part of sphinxtrain distribution.

Make sure to add the `-samprate 8000` option to the above command if you are decoding 8 kHz files!

The script `word-align.pl` from Sphinxtrain will report you the exact error rate which you can use to decide if the adaptation worked for you. It will look something like this:

```
TOTAL Words: 773 Correct: 669 Errors: 121

TOTAL Percent correct = 86.55% Error = 15.65% Accuracy = 84.35%
```

```
TOTAL Insertions: 17 Deletions: 11 Substitutions: 93
```

To see the speed of the decoding, check pocketsphinx logs, it should look like this:

```
INFO: batch.c(761): 2484510: 9.09 seconds speech, 0.25 seconds CPU, 0.25 seconds
wall

INFO: batch.c(763): 2484510: 0.03 xRT (CPU), 0.03 xRT (elapsed)
```

with `0.03 xRT` being a decoding speed (“0.03 times the recording time”).