CS 288: Statistical NLP
Assignment 2: Speech Recognition

Due September 29, 2014 at 5pm

Collaboration Policy

You are allowed to discuss the assignment with other students and collaborate on developing algorithms at a high level. However, your writeup and all of the code you submit must be entirely your own.

Setup

You will need:

1. `assign_speech.tar.gz`
2. `data_speech.tar.gz`

As with Project 1, `assign_speech.tar.gz` contains a library jar, a build script, and a skeleton implementation for where your code will go.

Description

In this project, you will implement a decoder for an automatic speech recognition system. Your decoder must take a sequence of MFCC vectors corresponding to an utterance and return its best guess at the words that were uttered.

The interface to implement is `edu.berkeley.nlp.assignments.assignspeech.SpeechRecognizer`; an instance of the implementing class should be constructed and returned in `SpeechRecognizerFactory`. You are given three components to integrate into your decoder. First, you have an acoustic model, which is a mapping from subphones to mixtures of Gaussians that can return emission probabilities for MFCCs being generated from each possible subphone; see below for more details on the acoustic model. Second, you have a pronunciation dictionary, which maps English words to sequences of

phonemes. Third, you have an n-gram language model as in Project 1. You are free to use your language model from Project 1; however, you can also use our reference implementation (bytecode included) as follows:

```
Iterable<List<String>> sents = SentenceCollection.Reader.readSentenceCollection(lmDataPath);
NgramLanguageModel lm = KneserNeyLmFactory.newLanguageModel(sents, false);
```

Be careful not to pass -1 to the language model; use `addAndGetIndex` instead of `indexOf` in the `Indexer` to make sure that OOV words are handled appropriately.

Because speech recognition has a complex search space, you will be unable to do exact decoding. You will need to implement beam search or another approximate search technique as discussed in lecture.

Acoustic Model The acoustic models the acoustic properties of context-dependent subphones. This is similar to the approach described in lecture, except that only the first subphone of each phoneme depends on the previous phoneme, only the last subphone depends on the next phoneme, and dependencies do not cross word boundaries. For example, we would map the phone sequence for "Berkeley" as follows:

B ER K L IY $\rightarrow$ B_1 B_2 $\text{B}_{3,ER}$ $\text{ER}_{1,B}$ ER_2 $\ldots$ $\text{IY}_{1,L}$ IY_2 IY_3

We represent context-dependent subphones with the `SubphoneWithContext` class: each instance stores the current phoneme (B, ER, etc.), the current subphone position (1, 2, or 3), and possibly a previous phoneme context or a next phoneme context, which may be the empty string. For example, $\text{B}_{3,ER}$ has ER as its next phoneme context (and no previous context), and IY_3 has the empty string for both contexts (because it is at the end of the word).

The acoustic model takes a frame and a context-dependent subphone and returns a log probability based on marginalizing over the components of a Gaussian mixture. Letting s denote a context-depending subphone, v denote an MFCC vector, and c denote a mixture index (with k total mixture components), we have:

$$p(v|s) = \sum_{c=1}^{k} p(c|s)p(v|c)$$

as the probability returned by the acoustic model. You can treat this component as a black box.

Note that subphones generally last for more than one frame, so you will want your model to be able to stay in a given subphone for multiple frames. We do not impose transition probabilities between phonemes, unlike the system described in Jurafsky and Martin, where the probability of staying on the current subphone versus transitioning to the next is modeled explicitly. However, you will still need to take the language model into account when transitioning between different words.

Pauses have been filtered out from the input data. However, the acoustic model still has a state corresponding to silence (`SubphoneWithContext.silenceState`).

Pronunciation Dictionary The `PronunciationDictionary` maps from words to sequences of phonemes. It exposes two methods: `getContainedWords`, which returns the set of words (all lowercased) that are included in the dictionary, and `getPronunciations`, which gives the pronunciations for a specific word. This gives you a way of mapping from sequences of phonemes to words and constraining your model to only produce actual words. The pronunciation dictionary may imply context-dependent subphones that were never observed in the training data and thus which the acoustic model has no mapping for: you can either prohibit these in your model or assign them a constant log-probability.

Data Harnesses are included to read all of the relevant data. The `audio/` directory contains wav files for 200 utterances by a single male speaker reading sentences from the Wall Street Journal, with one sentence per file. Each file also has a corresponding set of MFCCs, represented in text. These are 39-dimensional MFCCs extracted over 10 millisecond windows, with long silences in the audio files filtered out.[1] `test.txt` contains lowercased tokenized transcripts for each utterance in the audio. `wsj-and-train-lm.txt` is a small language model training set aggregated from Wall Street Journal data (which the corpus was drawn from) as well as the training set from which the acoustic model was trained. `cmudict.0.7a.txt` is a copy of the CMU pronunciation dictionary, which maps words to phonemes. Each line is a capitalized word followed by a sequence of phonemes in the "Arpabet." Finally, `acoustic-model.txt.gz` is a set of parameters for an acoustic model trained on data from the same speaker as in the test set.

Evaluation Your speech recognition will be evaluated both on the basis of decoding speed and the word error rate (WER) it achieves on the 200-sentence test set. Your code must run from start to finish in less than an hour and use less than 2G of RAM, but other than that we are not giving you hard requirements in terms of performance. To give you a sense of scale, a high-performing system should be able to get below 50 WER and should decode the test set in less than an hour. It should also fit in less than 2G of RAM. Our reference implementation gets 34.6 WER in 37 minutes.

You should strive to do something cool beyond a basic beam search implementation; see Implementation Tips below and the Aubert paper for some ideas.

When we autograde your submitted code, we will run the following command:

```
java -cp assign_speech.jar:assign_speech-submit.jar -server -mx2000m
  edu.berkeley.nlp.assignments.assignspeech.SpeechTester
  -path path/to/data_speech/csr-288
```

Implementation Tips The main challenge of this assignment is thinking about the complex state space of the search problem for speech recognition. What information do you need to maintain in

[1] This means that your decoder shouldn't have to do anything fancy to account for silences as a result of pauses by the speaker.

your states? What are the possible successors for a given state? How will you recover the sequence of words? You might try incrementally: first, just try to find the most likely phone sequence, then constrain the search such that only valid words are generated, then integrate the language model.

You will probably want to experiment with exponentiating your language model, i.e. multiplying the log probabilities it returns by a constant; this trades off the strength of the acoustic model with the strength of the language model. Also you may want to incorporate a "word bonus" to prevent the decoder from preferring hypotheses where very few words are generated.

Beam search is typically implemented with object-heavy data structures like Lists and Maps, where nodes in the search tree are explicitly represented as objects. This is sufficient for completing the assignment, though you may be able to use primitive types and do it substantially faster (however, you should probably build a simpler implementation first so you have something to debug against). That said, you will want to make sure that the nodes in your search tree are both a) relatively fast to build, since you will be generating and scoring lots of successor states, and b) relatively small, not only to save memory but also to save the time you would spend allocating memory. By optimizing your beam search, you should be able to use larger beams and see improvements in WER.

Moreover, beam search is typically done without a heuristic. Provably admissible and consistent heuristics can be difficult to devise, but often even coarse estimates of completion costs can substantially improve the quality of hypotheses returned. Feel free to experiment with things that come to mind! Two possible ways of integrating future costs are so-called language model smearing, in which language model scores are applied to sequences of phones that form only partial words, and exact forward cost estimation in a coarse model. Obviously, these can take time to compute, which reduces the size of the beam that you can use, but they can improve the search enough that they easily pay for themselves.

Submission and Grading

Write-ups You should turn in a 2-3 page write-up following the guidelines in Project 1. Since you have a significant amount of freedom in how you implement the decoder, you should clearly describe what implementation choices you made, along with reporting your performance on the various relevant metrics (WER, decoding time, etc.) and providing error analysis.

Submission You will submit `assign_speech-submit.jar` and a PDF of your writeup to an online system. We will sanity-check with the following command, which tries to decode one sentence:

```
java -cp assign_speech.jar:assign_speech-submit.jar -server -mx300m
  edu.berkeley.nlp.assignments.assignspeech.SpeechTester
  -path path/to/data_speech/csr-288 -sanityCheck
```

Again, this is primarily to check that your jar has built correctly; don't worry if you get an error from allocating too much memory.