Prosody Principal Components Analysis (PPCA) Workflow Notes

Nigel Ward

August 5, 2014

Abstract: Applying Principal Component Analysis to prosodic features has been useful for many tasks. This report describes the steps and software we use to do it.

Specifically it refers to version xxx, dated yyy, uploaded to the web on zzz.

Unless otherwise specified, everything is locally in linux-side directory `/home/research/isg/speech/ppca/`, including this file, as `ppca.tex` in `doc/`.

1 History

Early in 2011 Olac Fuentes suggested we avoid the problems of independently conditioning on non-independent prosodic features by applying principal components analysis. In Summer of 2011 Justin McManus prototyped the use of PCA on prosodic features for language modeling, working with just four raw features. Starting Fall 2011, Alejandro Vega extended the code to handle more features, in particular, making it work for features at different offsets and over different window sizes, and documented it in "Principal Component Analysis on Long Range Prosodic Features", available locally at `/home/research/isg/speech/uteplm/documentation/howto.tex` and `/home/research/isg/speech/timelm/switchboardPCx/documentation/`.

Starting late 2012, I reimplemented most everything, in particular, I:

- separated out the PCA code from the language-modeling code
- made collation of features parameterizable
- enabled feature-crunching (and normalization) without rotation to produce input for machine-learning algorithms, for Karen and Shreyas
- documented everything

Improvements still pending include:

- making the whole workflow run on 64-bit Linux
- enabling the easy inclusion of new features (Section 6.1.1)
- supporting realtime computation on microphone input

2 Motivation and Basics

Please read Ward and Vega 2012 (Sigdial) and/or Ward 2014 (Speech Prosody) before tackling this document or the code.

Other work illustrating the use of (some version of) this workflow includes our Interspeech 2012 paper applying it to language modeling, Alex's Masters thesis (2012) evaluating lots of features, my paper on *uh-huh* at the 2012 Feedback Behaviors Workshop, Steve's papers on utility for audio search (Interspeech 2013, MediaEval 2013, Speech Communications submitted), and Karen's paper on predicting importance (Sigdial 2013).

Ongoing work includes: 1) use of the dimensions for speech synthesis, 2) examining patterns of learner behavior in terms of the dimensions.

These techniques could also be applied to: 3) examining differences a) among individuals in their use of the dimensions, b) in dimensions across dialog types, and c) in dimensions across languages, 4) discovering the dimensions of different languages, and 5) enabling other researchers to do some or all of these things.

This document is written for three audiences: people wanting to learn more about how this works, people wanting to get the code working for themselves, and people wanting to modify or extend the code.

3 Use Cases

There are two main use cases.

(There's a python script `pfrotate.py` (for "prosodic feature rotate") that in theory does everything, but in practice you'll probably want to run the two separately (in part because our Matlab only work on 64-bit linux and respond only on 32-bit linux).)

3.1 Apply Rotation

This computes, for a dialog, the principal components active at each moment. This creates a `.pc` file for each track of one or more .au files.

For most purposes this will be done using some standard, pre-computed principal components, together with some standard normalization parameters.

(The results may make more sense if the file to be processed is from the same set as the audio used to generate the normalization parameters (Section 3.2). In particular, things may get strange if domains, speaking styles, or languages are different. Recording conditions may also be an issue, although the raw features output by `respond` are somewhat robust to these.)

The steps involved are:

- compute and read in the features
- fancy-normalize them, using some precomputed parameters
- rotate them, using a precomputed rotation

3.2 Compute Rotation

In order to do the above, there of course needs to be a normalization-and-rotation available to work with.

The steps are:

- compute and read in the features
- compute normalization params, then use them to fancy-normalize the features

- compute the rotation, that is, do PCA to discover the dimensions
- save the rotation coeffs and the norm params for later use (Section 3.1)

3.3 Overview of the Arguments

In general, there are five things involved in specifying a PCA process:

tracklist file specifying which tracks to process, each being a track from an audio file. Specifies the directory, then for each track the filename and the channel (left or right)
featurespec file file specifying the set of features to use (Section 4.3)
feature dir directory where to store (or find) the temporary files (.ep and .if)
parameter dir directory where to store (or read) the params and coeffs, and the various human-readable files, notably the logfile, correlation coefficients, and factor loadings
output dir directory where to write the resulting `.pc` files (one per track)

4 File Types

4.1 Data Files

First there are the data files, each representing an audio track or file, at various stages of processing.

.au The input. An audio file in Sun format, specifically 16 bit, linear PCM, 8K sampling, stereo. These are typically created by applying sox to convert a .wav file.
.f0 a pitch file with a pitch point every 10 milliseconds
.ep an Energy-and-Pitch-feature file
.if an individual-feature file, described later.
.pc The output: a principal components file. There is no header. Each line describes the prosody at one point. Points are 10ms apart. Each line contains a whitespace-separated list of the values for all the principal components, in order of variance explained. This has the same format as the `.ep` files.

Note that all but the initial .au file can be recomputed, so they don't need to be saved. In particular, the `.ep` and `.if` files are intermediate files that have no independent value, so need not be kept, except possibly to save computation time.

4.2 Tracklist Files

This specifies the audio tracks to process. The first line is the directory in which the audio files are located. (Old files lack this and it needs to be added.) Subsequent lines specify the track and the file. For example the line

```
l sw02079.au
```

means to process the left track of the specified Switchboard audio file. Tracklist files have the extension `.tl`.

A standard tracklist, which is probably the same used in our Sigdial 2012 paper, is `fulltest/alex16.tl`.

Tracklists are used to directly specify the `.au` files to compute features for, for `multirespond.py`. They are also used to partially specify the feature files to use for `createnormrot.m` and `.applynormrot.m`; in those cases the directory to use and the extension to use are different, the former is passed in as a parameter, and the latter is hardcoded.

4.3 Feature Specification Files

To encode contextual information we need to use features computed at various temporal offsets, relative to the point of interest. A "featureset specification" (`.fss`) file specifies which features to use. These are sometimes called "crunch" files since they describe how to crunch together data from multiple feature files into a single composite file suitable for machine learning or dimensionality reduction.

In general, it's probably fine to use one of the standard feature specifications, such as:

minitest/minicrunch.fss , 11 features, for testing the workflow
social/symmetric.fss , 96 features, used for social speech
fulltest/fullcrunch.fss , 112 features, too many to use except on a big-memory machine
fulltest/slim.fss , 78 features, 48 self, 30 interlocutor, the new standard set

In a `.fss` file each line specifies a feature, a window size, and an offset, for example

```
pitch-100 -200 self
volume-200 400 interloc
```

where the first line means the speaker's average pitch over a 100ms window that starts 200ms before the point of interest, and the second line the interlocutor's average volume over a 200ms window that starts 400ms after the point of interest.

Since `getfeaturespec.m` parses this by fixed offsets, it's important for everything to line up exactly, and to use spaces not tabs.

`.fss` files are used specifically in:

multirespond.py , to determine what window sizes to invoke `respond` with.
load_features.m , to determine which features to read in and assemble.
fetch_and_preen.m called by the above
getfeaturespec.m , which is the subroutine that actually parses the features, called in many places

4.4 rotationspec.mat

This is a matlab file that contains the information pertaining to a rotation. This enables the application of an pre-determined rotation to new files. It contains

- the normalization parameters, namely for each feature its mean and its standard deviation
- the PCA coefficients

A related file is `loadings.txt`, which is a human-readable version of the PCA coefficients.

5 File Organization

It's probably best to create a new directory for each project. If all relevant Matlab work is done in this directory, `rotationspec.mat` will be written here and then found again without difficulty. This directory will also naturally house the various human-readable files and the logfiles. The `.pc` output files can also sensibly go here.

However there's no need to store the temporary files in this directory. Since currently matlab and respond don't run on the same machine can't use `/tmp` for this. A handy place to use for this is `epPark`; this is hardcoded in `multrespond.py`.

6 Code

Figure 1 overviews the components.

6.1 Basic Now-Feature Production

PCA starts with some collection of raw features. These are generated in two steps, both conveniently done by the python program `multirespond.py` (Section 6.2).

Multirespond first calls a C program, `respond`, to do the actual signal processing. `respond` program takes a `.au`-format stereo file and writes two `.ep` files, one for each track. It does this for four base features: volume, pitch range, pitch height, and frame-by-frame amplitude variation ("ampvar"), as a proxy for speaking rate.

The working version of respond is in `uteplm/alex_aizula_test/out/`.

Internally `respond` first does some preparatory work: computing the raw energy, and invoking `f0calc` to compute the raw pitch. Second it normalizes the energy, to overcome any recording-condition differences in average speaking volume and in average noise level. To do this it finds the typical-silence and typical-speech values of energy, and then normalizes the energy with respect to these values. Third, it cleans the pitch values and converts them to percentiles. This normalizes for individual differences in pitch height and in pitch range.

Finally it computes the four features. These can be computed over windows of various widths, specified as arguments. Each window is "anchored" by its right edge, thus in the output the feature values for time t are actually the values computed over windows ending at time t. If a window extends beyond the start of a file, then the value is `-1`, thus flagged as invalid. Similarly, there is a `-1` flag for windows in which the pitch height cannot be computed because there are no pitch points, and for windows where the range cannot be computed because there are less than two pitch points.

6.1.1 Other Potential Now Features

Other raw features may later be added. For example this might include features are generated by `Praat` (notably NHR) and by `mrate` (namely speaking rate, although in our Specom 2012 paper we found it worse than amplitude variation (ampvar, sometimes also called jitter) as a speaking-rate proxy).

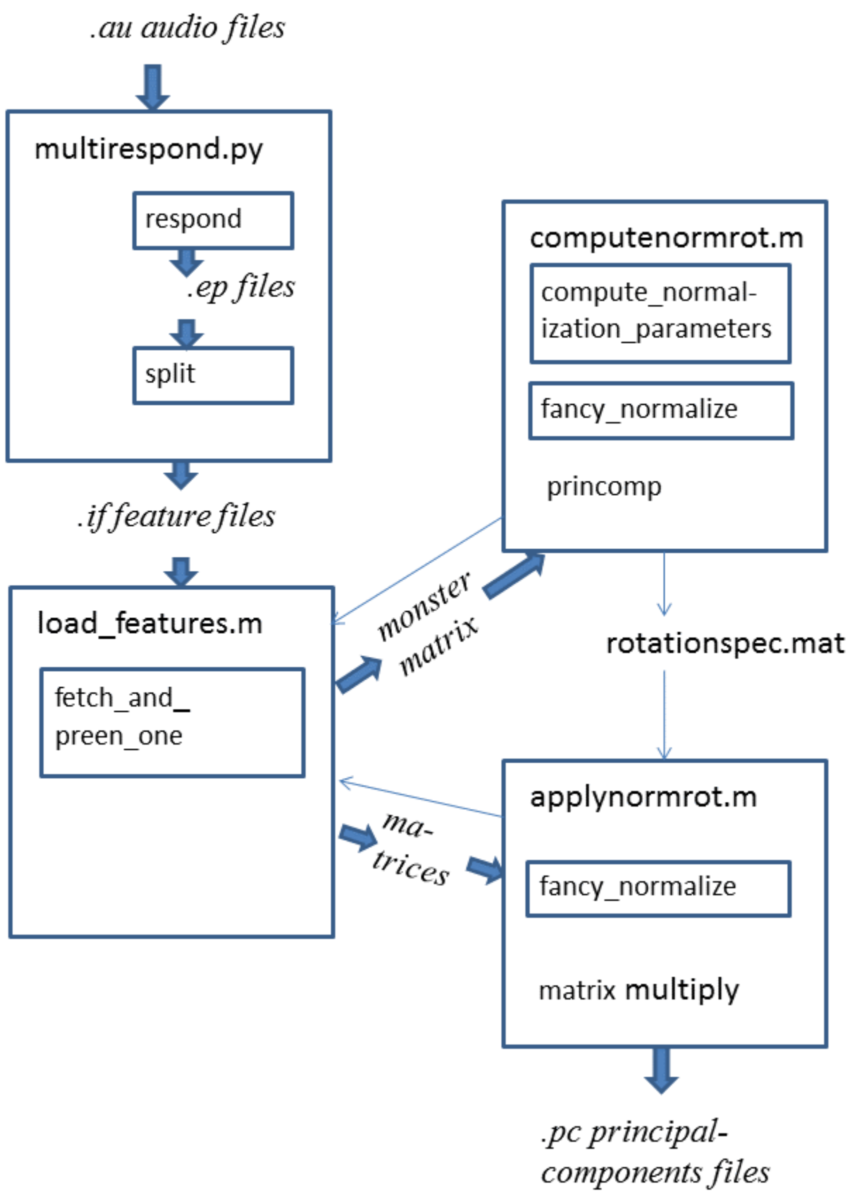

Figure 1: Workflow Overview

Respond could also be extended to generate a voicing-fraction feature and maybe a speaking-fraction feature, to say for each window in which fraction of the frames there was voicing, respectively, speech.

For the future discovery of visualization-friendly dimensions, we'll want to use filterbank features, for example percent of frames in this window where the sound fell in the 80–100%ile energy range, percent where it fell in the 0–20%ile pitch region, etc. This would also obviate the need to patch in average pitch values to replace non-existing pitch points. This would also enable a nice handling of the Barnes effect (Jon Barnes, Speech Prosody 2014 etc.), that the perceptual significance of pitch points is proportional to the volume at that moment.

Note that NHR and the filterbank features only take on non-negative values, something to consider when designing normalization.

6.2 Computing Features over Various Widths

Each invocation of `respond` only gives results for a single window size of each type, thus only one pitch-height feature, one pitch-range feature, etc. Thus to produce windows of various sizes, we need to invoke `respond` multiple times.

(Alex had a different way to generate values of windows for multiple sizes. He used the code in timelm/switchboardPCx/javaScripts/PCAJava/ to produce mock longer-window estimates by just averaging the values for adjacent small windows. While this seemed to work fine, it's not the obvious way to compute pitch height and pitch range over larger windows.)

`multirespond.py` does this. It two arguments, a tracklist file, and a featurespec file, as described above. Thus for example, from `src/`:

> python import multirespond.py multirespond.everything("../minitest/minitracklist.tl", "../minitest/minicrunch.fss")

First, from the featurespec file, it infers what the window sizes it needs to generate. For example, it might discover that it needs to produce features over windows of sizes 50, 100, 200, 400, 800, and 1600 (all in milliseconds).

Next it invokes `respond` for each track and each window size. This produces oodles of `.ep` files, which are all written to feature directory, traditionally `epPark`.

To make this information easier for matlab to parse (specifically `load_features.m`), multirespond then "splits" these to create individual-feature files `.if` files (formerly `.sf files`), each containing has the data for only one feature. The result is multiple oodles of files with names like `sw2001-l-ph-100.if` and `sw2001-r-pr-200.if`. These also go in the feature directory. To save space, these do not contain timestamps; they have just one value per line.

This step is not quite trivial because `.ep` files lack values (and even lines) for the first few frames. This is because, for example, when `respond` is asked to compute features with window size 50ms, it outputs no data for the first 5 frames, so the `.ep` file starts at 50ms. As part of the splitting process, `multirespond` therefore pads the start of each `.if` file with the appropriate number of lines, using -1 (invalid) as the value there. For this it uses `longpadding.txt`. Thus the `.if` files end up all aligned properly, all starting at time 0.

Note that `importdata` appears to be taking much longer than is reasonable, and other Matlab users have noted this too, with no solution suggested, alas.

6.3 Feature Assembly/Collating ("TimeCrunching")

The relevant features at any point in time are not just those anchored at that point, but also contextual features from the past or future, and from the interlocutor as well as the speaker. We therefore need to asemble all these features from the various sources (the various `.if` files.

(This was previously handled by Alex's `timelm/switchboardPCx/pScripts/genXtraFeatures.py`, with the parameters hard-coded, plus a separate step to fold in the interlocutor-track features, using `timelm/switchboardPCx/pScripts/padinterlocutor.py`.)

This is done in Matlab by `load_features.m` which assembles everything specified in the `.fss` file. The output is a huge array with *nfeatures* columns (some shifted it up or down in time according to the specification) and *ntimepoints* rows. This output is used directly by the calling program, either `createnormrot.m` or `applynormrot.m`.

Local note: Matlab currently runs only on the 64-bit machines, e.g. `lisa`, so be sure to ssh there.

If one wants to just write the crunched features, there's an option here to do so, currently commented out. This writes the features as a huge `.crf` file (for crunched-raw-feature file), which one might then use for machine learning. "Raw" here simply means "not rotated by PCA".

6.4 Examining the Correlations

One can examine the correlations among the features. This can be for general edification or as an indirect check on correctness of the feature computation and collating. In particular we can look for and examine the most highly correlated few columns, and the least highly correlated few.

To make the correlations readable, we map them back to the names of the features involved. This is done by `output_correlations.m`, which is called from `load_features.m` and writes a human-readable file, `pre-norm-corr.txt`. Post-normalization correlations are also available; these are written by `computenormrot.m`.

6.5 Normalization

PCA requires that we normalize the features. It's important that they have zero-mean, and also that they have approximately the same standard deviations.

Note that we do *not* normalize by file. Any particular speaker may have his own typical speaking style, and we don't want to kill that information (although some has already been removed by the time we get here, by the normalization inside `respond`, as discussed above). Thus we normalize over all the files in some large set; the same large set of data that we'll use for the PCA.

This is done by `createnormalrot.m` where first reads the features using `load_features.m` then normalizes them with `compute_normalization_params` and then `fancy_normalize.m`. It then continues on to do PCA (Section 6.6).

(Alex's normalization code was `timelm/switchboardPCx/pScripts/normalizeEP.py`.)

6.5.1 Thoughts on Normalization

Correct normalization is a huge issue.

Here the purpose of normalization is just to make sure the features all have the same standard deviations, so that none unnaturally dominates the PCA. So z-normalization is fine, and it doesn't matter that the incoming distributions are largely not normal. (Pitch height, being percentile-based, is flat in distribution. Energy is bimodal, with 0 being typical silence and 1 being typical speech. ampvar is probably unimodal, with high values in consonant-dense regions, moderate values in slower regions, and small values in silence regions. The distribution of the pitch range values is probably skewed, with lots of values near zero and a long tail.)

Now to comment on each feature in turn:

- Volume. Volume varies enormously across tracks, mostly due to differences in recording conditions. Normalizing for this is our main concern. The average volume across tracks will, of course, vary with the amount of speaking the person in that track is doing. Thus we want to ensure that each person, when he is speaking, is reported has having the same average volume. This is of course not true, since some people have quieter voices than others, but we can't really detect that. Also that probably doesn't matter, since we're only interested, for most purposes, in whether a speaker is being quiet now relative to his typical speaking volume. There are also slow variations in gain, as the speaker holds the handset closer or farther over time; these we also don't deal with. The bottom line is, the values output by respond are already track-normalized, and should not be messed with. Of course, we can normalize over an entire set of dialogs to bring the overall mean to zero, for example, but when Shreyas normalized, file-by-file, to have each individual file have zero mean, then all language-modeling benefit was lost.
- Pitch Height. Pitch height will be essentially robust to the recording conditions. However it is, like volume, also very speaker-independent. By expressing it in percentiles, we have already compensated for this.
  Pitch features are tricky, both pitch height and pitch range, since we need to deal with the -1 flags. Here's a solution (not a great one): we compute the average value over the valid (not -1) points, then we substitute this average values for all the -1s. Then we happily z-normalize everything.
- Pitch Range. The correct way to normalize this is far from clear. Some pitch patterns, such as the minor third, should be normalized in terms of semitones (percent change in Hz relative to the starting point). However for other things, such as jump to a high pitch for emphasis, what types of variation are perceptually equivalent, is unknown. For now, this is also represented in terms of percentiles. This is adequate for distinguishing between monotone and everything else (including excited speech, swoopy speech, and specific pitch features such as sharp pitch drops).
- Speaking Rate. This is track-normalized in `respond`, although the quality of this has only been spot-checked.

6.6 Determining the Rotation (doing the PCA)

Done in `createnormrot.m` (create normalization and rotation).

This creates a set of normalization parameters and factor loadings, and writes it to `rotationspec.mat` for later use by `applynormrot.m`.

This first calls the code to load, preen, and normalize the features.

Then comes the PCA. This is done using Matlab's `princomp` function. (This is memory-intensive. The new Lisa has 32 Gb. Note that on a 4GB Alex's machine can only handle 600K datapoints with 76 features, or 450K with 103 features. Might downsample to every 20ms to be able to handle effectively more data.

6.7 Examining the Dimensions

After the PCA, we'll often want to examine the variance explained by the dimensions, and the cumulative variance explained. This can be done by loading the `rotationspec.mat` file and then applying:

```
load rotationspec
latent ./ sum(latent)
cumsum(latent) ./ sum(latent)
pareto(latent ./ sum(latent))   # produces a cool graph
```

We will also often wish to analyze the dimensions. `createnormrot.m` includes a call to `writeloadings.m`, which takes as input the `coeff` matrix returned by PCA, and the feature-name information assembled by `getfeaturespec.m`. Each column of coef represents the feature loadings for one dimension. For each, we output a list of featurename-loading pairs, as one large file, called `loadings.txt`, for example

```
dimension1  0.12 sel-vo-50+0
dimension1 -1.08 int-ph-400-200
dimension1  0.01 sel-pr+50+100
...
dimension2  0.67 sel-sr+0+100
...
```

Such a file can then be explored to find out things about the features involved in each dimension, for example, using the Unix `sort` and `grep` commands to find, for example, the features with the highest loadings, with the strongest (highest absolute) loadings, the volume features with the highest/lowest loadings (via `grep`), and so on.

For example,

```
sort -k2n xxx.pc
awk '{print $2*$2, $0}' xxx.pc | sort -n | cut -d " " -f2-
```

To truly understand a dimension, we also will want to find datapoints high/low on that dimension and listen to what's happening in the dialog there. This is possibly after applying a rotation to generate `.pc` files, as described in the next section.

6.8 Rotate

This is done by `applynormrot.m`, which applies a previously saved `rotationspec.mat`, namely the one found in the current directory.

This and the previous step are packaged up as `twostep.m`.

6.9 Examining Variation in Uses of the Dimensions

For each dimension, we'll want to examine individual variation and (somewhat later) group variation.

The between-groups comparison will compare all learner data with all native data, in terms of the two summary statistics, to find out which dimensions they differ on.

The two summary statistics are:

- average value (to detect bias to one side of the dimensions)
- average absolute value (to detect failure to use a dimension much)

In addition, it would be nice to generate histograms for each dimension, including superimposed histograms for the two populations.

This is done by `write_summary_stats.m`, whose input is the rotated matrix. For each column of the matrix (each feature), we compute these things. This is called by `applynormrot.m`.

6.10 Interpreting the Dimensions

To understand the dimensions, various things can be done.

6.10.1 Examine the Factor Loadings

Look at the numbers. A useful thing is first to look at the volume features (with `grep`) for the "self" speaker, to find out when they're talking. A useful next step is to look at the "interloc" volume features. Next it's useful to use `sort` to find the strongly negative and strongly positive factors on this dimension.

a. visualize them, using the text file `loadings.txt` as mentioned above. Also the code in `~/papers/dublin/pattern.m` and `ppattern.m`

Old factor-loading examples are in `isg/speech/timelm/switchboardPCx/factorLoadings`, generated by `switchboardPCx/factorLoadings.py`.

6.10.2 Listen to Extreme Examples

To help understand a dimension, it's helpful to listen to locations where each dimension has extreme (the highest and lowest) values.

This is done with `find-extremes.py`, which takes two arguments, the directory of the `.pc` files to process, and the number of dimensions to process. The results are written as textfiles to the `extremes` subdirectory.

In the past, these extreme points were found using the `switchboardPCx` version of `find-extremes.py`. Some timestamps of extremes on Alex's analysis are in `isg/speech/timelm/switchboardPCx/audioExamples`, and audio clips for those are in `/home/users/nigel/papers/dimensions/snippets`.

One issue is that adjacent timepoints typically have similar values. So there ought to be a pruning stage where we drop all points within 500ms of a more extreme point, either in matlab, or in python (perhaps by a call to Unix's uniq).

Generally we want the absolutely most extreme points, across all the files, but may also sometimes want to find one extreme point per file, for some diversity.

Once we have these timepoints, it's time to listen. There are lots of tools that can do this, but we want one that can easily let you jump to 5 seconds before this point, then play this region. Invokable from the command line is a big plus. Using second notation (not minutes and seconds) is also nice. Dede used to do all these things, but no current machine seems to run it.

To Do:

- to write dede calls to play these and
- sox calls to clip out snippets of these and optionally add a a "beep" to mark the precisely the most extreme point in that clip
- python calls to lookup the words said at those points

Also to do: find locations where each dimension has values that are high and low *relative to other values*. This is important because the highest instance of dimension x may also be high on dimension y, and the effects of y may mask those of x. **for now, eyeball some high/low locations in the .pc file, to figure out how best to handle this. Some preliminary thoughts:

- In matlab, find the norm (the mean absolute value) for every feature.

- Divide all feature values by their feature's norm.

- For each feature, define its "salience" at each point as the ratio of that feature's absolute value to the max of the absolute values of all other feature.

- output the filename, the timepoint, the normalized value for the feature of interest, the salience, and the name of the strongest rival feature

Any files can be used for extreme-based dimension interpreting. If doing Switchboard, it's nice to work with the files examined for the Sigdial 2012 paper, for the sake of familiarity. These are listed in `fulltest/alexscan.tl`

The Switchboard audio is in `/isg/speech/uteplm/switchboardau/` with a longer set in `timelm/switchboardau/aufiles` . You can listen to it using `/home/research/isg/speech/workingDede/dede`. If dede crashes, copy `/home/research/isg/speech/workingDede/piau-au-file.PCM` to `/tmp` and restart it.

6.10.3 Consider Co-occurring words

find which words co-occur with values high/low on each dimension.

The old listings are in `switchboardPCx/countFiles/sratios`.

Timing

Multirespond on Abe takes about 15 seconds per 5-minute audio file with 3 window sizes; more if the .f0 files need to be computed.

Creatnormrot on Lisa takes about 60 seconds per 5-minute track, almost all of it just loading the feeatures. 40 minutes to do a 78-feature load on about 60 track-minutes.

4 minutes to write a 10-minute .pc file.

To Do

Quick Stuff

- write a logfile, including timing information

Workflow completions

- support for listening

- support for visualization

- support for discovering word-dimension tendencies

Bugs and Speed-Ups

- alter multirespond.py to downsample to half-size the .if files i.e., make an interval of 20 ms between samples

- add matlab code to read in a .if file and save it as an equivalent .mat file, to save time on subsequent reads. Might implement as a form of memoization/caching.

- edit respond so that it can be run remotely; the key is to change it so it doesn't try to doesn't grab the audio device

- time the various steps, especially the PCA, as a function of size, on various machines (32G new Lisa, for example)

L2 paper stuff

- run on Alex's training data, compare the resulting dimensions with his (look at feature loadings, extreme-valued datapoints) [8]

- try on Spanish or Japanese [40]

Futuristic stuff

- try with filterbank-style features

- measure robustness of derived features to variations in training set; software to match up a dimension in one rotation to the most-aligned dimension in another rotation (similarity metric is `sqrt(sum((a-b).^2))`);