Midlevel Prosodic Features and Prosody Principal Components Analysis (PPCA) Toolkit
Version 5

Nigel Ward, University of Texas at El Paso and Kyoto University

December 21, 2015

1 Overview

This toolkit supports prosodic analysis of speech, especially dialog, and especially statistical and machine-learning work. This may be useful for you:

- as a source of code to compute individual prosodic features
- for producing a full set of prosodic features suitable as input to machine-learning algorithms
- as a complete workflow for discovering patterns of prosodic or multimodal behavior.

It contains a a novel set of prosodic features that are robust, everywhere-computable and fairly comprehensive. It also contains a workflow for analysis including Principal Components Analysis (PCA) and various support for automated and human-in-the-loop analyses.

2 Getting Started

The code is at `https://github.com/nigelgward/midlevel` . It requires:

- Matlab (tested on release r2015a, but reported to also work, with minor modifications (e.g. a download of a freeware version of strsplit), on earlier release)
- Mike Brookes' Voicebox code, for audio file input and the pitch computation, available from `http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html` .

To see if it runs for you, enter Matlab, do addpath for midlevel/src4 and voicebox, cd to the flowtest subdirectory, and do `findDimensions('singletrack.tl', 'minicrunch.fss');`. In a minute or so this should create a loadings file; if so, you should be good to go.

If you are only interested in using the features, skip ahead to read Section 7.

In general, to prepare to work on your own data, you'll need to do three things: 1) convert it to the right format, 2) create a file inventorying your audio, and 3) select a feature set to use. Section 4 has more information.
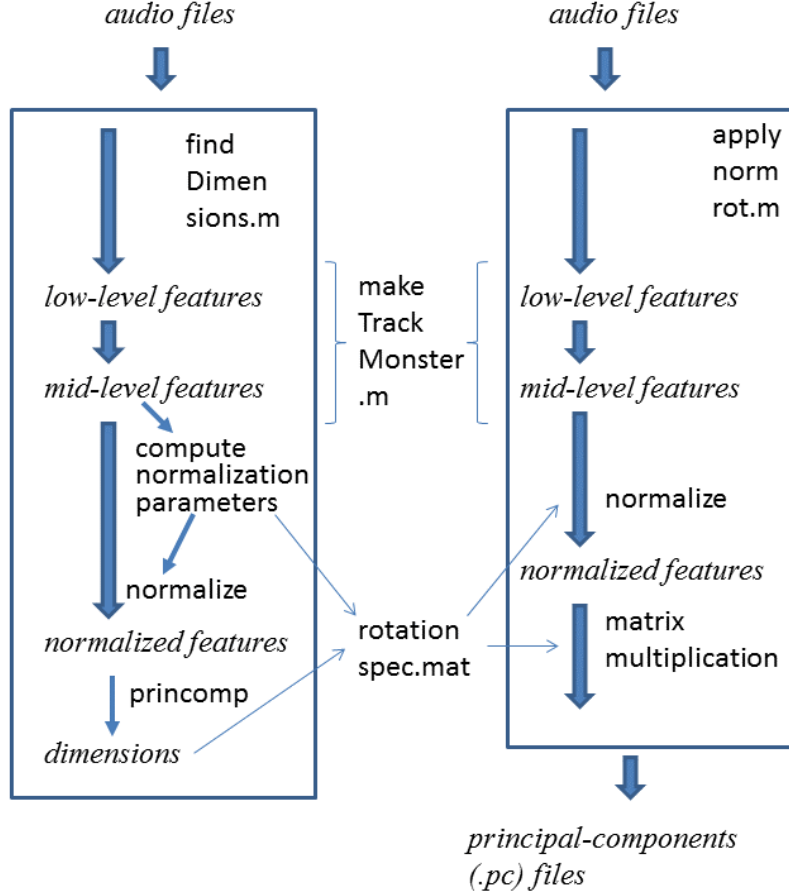
Figure 1: Workflow Overview

3 PCA-Based Analysis Overview

This toolkit was designed to apply Principal Components Analysis over large sets of prosodic features over tens of minutes of dialog data. This is useful, we have found, for various purposes. It gives dimensions which correspond to interpretable patterns of behavior [1]. The values of these dimensions usefully characterize the instantaneous state of the dialog [2]. Applications so far include language modeling, information retrieval, filtering, gaze prediction, distributional analysis, predicting actions from prosody, and examining non-native dialog patterns [3, 4, 5, 6, 7, 8]. Before reading on it's probably best to look at one of these papers for an overview of the approach and the features.

There are two main use cases, related as seen in Figure 1, and described in the following subsections.

3.1 Apply Rotation

This computes, for each moment of a dialog, the values of the principal components at that moment. 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), thus avoiding potential problems due to different domains, speaking styles, or languages. Recording conditions may also be an issue, although the features are designed to be somewhat robust to these.)

Thus the Matlab function `applynormrot.m` creates a `.pc` file for each track of one or more audio files. The steps are:

- read in an audio file or set of files
- compute the raw features
- normalize them, using precomputed parameters (means and standard deviations)
- rotate them, using a precomputed rotation
- write the results to `.pc` files.

The resulting dimension-based representation can then be as input to for various tasks, or can be interpreted, as described below.

3.2   Compute Rotation

In order to do the above, there of course needs to be a normalization-and-rotation available to work with. `findDimensions.m` creates this. The steps are:

- read in an audio file or files
- compute the raw features
- compute normalization params, then use them to normalize the features
- compute the rotation, that is, do PCA to discover the dimensions
- save the rotation coeffs and the normalization params for later use (Section 3.1)

The PCA itself is done using Matlab's `princomp` function. This is memory-intensive. If you're processing a lot of data, then the rotation should be computed on just a subset, then applied to it all. If this is not done, Matlab will gobble memory and freeze or crash the machine.

3.3   Overview of the Arguments

In general, five things are needed to completely specify either of the above processes. Three of these are arguments:

tracklist specifies which tracks to process, each being a track from an audio file (Section 4.2)
featurespec file specifies the set of features to use (Section 4.3)
output dir specifies where to write the resulting `.pc` files (one per track), and extremes files (one per dimension)

The other two things are locations which are set implicitly:

pitch cache is the subdirectory where to store (or find) the `fxrapt`-output f0 values, as `.mat` files. This subdirectory is created in the same directory where the audio files are located, as specified in the tracklist file.

parameter dir is the directory where to store (or find) the params and coeffs (notably in the file `rotationspec.mat`), and the various human-readable files, notably the logfile, correlation coefficients, and factor loadings. This directory is implicitly set to be the location where the matlab process is run.

Given the implicit storing of the params and coeffs, it's probably best to create a new directory for each project. If all relevant Matlab work is done in this directory, then all the parameter files will be written here and then found again without difficulty.

4 File Formats

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. A stereo audio file, 16 bit, linear PCM, 8K sampling, of no more than 10 minutes. While sometimes longer files, other .au formats, and even `.wav` files, work fine, it's safer to use this format. The `sox` program is convenient for conversions. The workflow is designed to handle stereo recordings, with each speaker in a separate track, but monoaural data can be handled by first simply converting it to a stereo file with the same data in each track.

—**f0.mat** a file specifying for an audio track the pitch every 10 milliseconds. These files are created because fxrapt is slow, so it's worth saving the results to avoid needing to later recompute them.

—**.pc** The output: a principal components file. There is a one-line header describing the provenance. Each subsequent line describes the prosody at one timepoint. These are 10ms apart. Each line contains a whitespace-separated list of, first the timepoint, then the values for all the principal components (PCs). PCs appear in order of the variance explained. These files are large and writing them takes a long time, so this function is currently commented out.

4.2 Tracklist Files

This specifies the audio tracks to process. The first line is the directory in which the audio files are located. 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`.

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 thus describe how to "crunch" together data from individual computations into a single composite matrix suitable for machine learning or dimensionality reduction.

Various `.fss` files exist, including `fulltest/al.fss`, an "assumption light" new set of mid-level features, including about 168 features, and `comparisons/april.fss`, which has more features for the primary-track talker than for the other talker.

In a `.fss` file each line specifies a feature, window start, window end, and the channel, for example

```
vo -200 -100 self
cr 400 600 inte
```

where the first line means the speaker's average volume over a 100ms window that starts 200ms before the point of interest, and the second line the interlocutor's average creakiness over a 200ms window that starts 400ms after the point of interest. "Self" refers to the primary track, left or right, as specified in the tracklist file, and "inte" to the other track, with the interlocutor's voice.

In these files currently the following codes are recognized:

New two-letter codes:

vo	intensity/volume
sr	speaking rate proxy
cr	creakiness
fp	flat pitch: degree of flatness
np	narrow pitch range: degree of narrowness
tp	typical pitch range (not often used)
wp	wide pitch range
hp	high pitch (obsolete)
lp	low pitch (obsolete)
th	true high pitch: degree of highness
tl	true low pitch: degree of lowness

Reserved two-letter codes:

sf	speaking fraction
vf	voicing fraction
p0...p5	pitch bands, as a potential replacement for lp and hp
sl	slowness, to replace sr
fa	fastness, ditto

Adding a new prosodic feature requires changing three things. First you create an entry for your new feature in the featurespec file, choosing any convenient two-letter code and an appropriate window size and offset. Second, you write a new matlab function to compute that feature. This might compute it from the audio, or from multimodal data, or it might read values from a file that was written by an external program. Third, you add a new case to the big switch in `makeTrackMonster` to associate your new feature-computing function with the two-letter code.

Every feature-computing function is responsible for returning a vector of values for windows centered every 10 milliseconds throughout the audio file. The first one is centered at 10 ms. This is true for both the frame-level features (energy and pitch) and for the derived (mid-level) features, which span longer windows. While the raw pitch features can include NaNs, this is not

appropriate for the mid-level features.

Thus all feature-computing functions must return values everywhere, even at the start and end of the audio file. Mid-level features have windows longer than 20ms, so windows centered close to the start or end of a file will stretch out beyond the point of no data, and thus they will lack enough information to return a well-considered value. In such cases the function should return other non-obtrusive value in the typical range (rather than some extreme value like -9999, since that would mess up the normalization). While sloppy, this is not a problem for now, since the audio files we work with are long enough (generally 5–10 minutes long) that the vast majority of features values will be valid.

4.4 Normalization and Rotation Parameter File

`rotationspec.mat` 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 Interpreting the Dimensions

An important reason to apply Principal Components Analysis is that it helps understand what's going on in the data. Indeed, for all data sets tried so far, the dimensions output by PCA turn out to be readily interpretable as patterns of behavior. There are three ways to do this interpretation, as described in this section.

First, however, you may wish to look at the variance and cumulative variance explained by the PCA-found dimensions. For this:

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

5.1 Examine the Factor Loadings

`findDimensions.m` includes a call to `writeloadings.m`, which writes a large, human-readable file called `loadings.txt`. While these files are readable, it's generally better to visualize the loadings. This can be done with:

```
diagramDimensions('rotationspec.mat', 'xxx.fss');
```

Where xxx.fss is the name of the feature file used to create the rotationspec. (If you don't remember, look at the value of `rotation_provenance` after you load the rotationspec. The only

reasons why you need to specify this by hand is that 1) rotationspecs written by earlier versions don't include the name of the fssfile, and 2) the fssfile may be in a distant directory.)

This creates a `loadingplots` directory and writes a bunch of .png diagrams there.

5.2   Listen to Extreme Examples

To understand a dimension, it helps to listen to locations in data where each dimension has extreme (the highest and lowest) values. To support this, by examining the files `dim00.txt` etc. in the `extremes` subdirectory of outdir. This is written by `findExtremes.m` (called by `applynormrot.m`). This finds the extreme points in each file, but winnows out points too close to each other, to provide some diversity.

Once we have these timepoints, it's time to listen. There are lots of tools that can do this. One option is "Didi" (http://www.cs.utep.edu/nigel/didi/), which conveniently lets you jump to 5 seconds before a desired timepoint, andthen play this region, and is conveniently invokable from the command line. However Didi only installs easily on 32-bit linux machines with Centos/Redhat 5.

5.3   Consider Co-occurring words

The last source of insight for interpreting the dimensions is to see find which words co-occur with values high/low on each dimension. Of course this is only possible if we have transcribed data, e.g. Switchboard. A workflow for this needs to be revived.

6   Comparing Populations

Different populations may use prosody in different ways. Examining their behavior with respect to the dimensions may be informative.

One way is to examine the summary statistics (`summary-stats.txt`) written automatically by `applynormrot.m`, for two different populations. The summary statistics are:

- average value (to detect bias to one side of the dimensions)
- standard deviation (to detect failure to use a dimension much)
- skewness
- kurtosis

Another way is to create histograms showing the distributions of two populations, on the various dimesions, and eyeball them. For example:

```
refvals = applynormrot('reference.tl', 'someset.fss', '/tmp');
newvals = applynormrot('new.tl', 'someset.fss','/tmp');
histograms(refvals, newvals);
```

7   Features

I am currently writing a full explanation of the features [9]; this section just notes a few points.

7.1 Frame-Level Feature Computation

The frame-level (low-level) features are computed: pitch and energy.

The pitch is done with `lookupOrComputePitch.m`, which is a wrapper for Mike Brookes's Voicebox function `fxrapt.m`; this gives values in hertz, or NaNs if there is no detectable pitch.

The low-level energy computation is done using `computeLogEnergy.m`.

Other frame-level features may later be added. For example this might include features generated by `Praat` (notably NHR).

If keystrokes are specified in the `.fss` file, `featurizeKeystrokes.m` is called to load that information; similarly `featurizeGaze.m` is called if gaze features are specified.

7.2 Track-Based Normalizations

Pitch is converted from hertz to percentiles, to normalize for individual differences in pitch height and in pitch range.

Energy is rescaled to normalize for individual differences and 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, using `findClusterMeans.m` and then normalizes the energy with respect to these values. This is done, not over the frame-level features, since those are probably too short, but as part of the subsequent energy-over-larger-window computations.

(This is not the simplest way to normalize, but it seems suitable. The average volume across a track will 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 volume on average. (Of course some people have quieter voices than others, but we're only interested in whether a speaker is being quiet or loud relative to his typical speaking volume.) There may also be slow variations in gain, if the talker varies the handset-to-mouth distance, but these we don't deal with.

7.3 Mid-Level Feature Computation

The mid-level features are as listed in Section 4.3. Each summarizes something about the values of the frame-level features across some window. The motivations for these specific choices of feature are in [9].

Each value is associated with the time at the center of the window. Windows are shifted (stepped) every 10ms, because it's unlikely that prosodic features change faster than that. Mid-level feature windows are always at least 50ms long, thus they are overlapped.

7.4 Feature Assembly

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 assemble all these features. Essentially this just requires concatenating the various mid-level features, shifted (offset) appropriately.

The output is a huge monster array with *nfeatures* columns and *ntimepoints* rows.

For some purposes these assembled features can be useful, as input to various machine learning algorithms, without going on to the rotation step. To write data for such purposes, one can add a call to `write_pc_file.m` on the monster array.

7.5   Overall Normalization

Before doing PCA we need to normalize the features to all have zero mean across all dialogs in the training set. (This is subsequent to the frame-level pitch and energy normalizations described above.) It's also helpful to normalize so that each feature has same standard deviations, so that features with larger variance do not dominate. (The mid-level features are far from normally distributed, and after normalization that's still true, but this is probably only an aesthetic problem.)

Note that we do *not* normalize by file. Any particular speaker may have his own typical speaking style, different from others, and we don't want to lose that information. (When Shreyas tried normalizing, file-by-file, to have each individual file have zero mean, all language-modeling benefit was lost.)

8   Validation

Testing for most of the feature computation methods was done using both synthetic test data and small audio test files. Details are given in the comments of each Matlab file.

To see the values of various low-level and mid-level features as they vary over an audio file, uncomment the various `plot` commands in `makeTrackMonster.m`. One can then listen to the audio file, using any available player, to see whether the feature values are indeed high and low where they should be.

As an indirect check on correctness of the feature computation and collating, one can examine the correlations among the features. Every call to `findDimensions.m` creates a file, `post-norm-corr.txt`, which lists, for each feature, the most highly correlated and most anti-correlated other features. (This is output by `output_correlations.m`.)

9   History

Version 1. In our language modeling work, we observed problems due to the non-independence of our prosodic feature set. Early in 2011 Olac Fuentes suggested we solve this 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.

Version 2. 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/`. He applied these to Switchboard data, probably the files listed in `fulltest/alex16.tl`. (The audio files are on the CDs, but some other sample Switchboard files are in `/isg/speech/uteplm/switchboardau/`.) The factor loadings this gave are in `isg/speech/timelm/switchboardPCx/factorLoadings`,

generated by `switchboardPCx/factorLoadings.py`. Extreme examples for each dimension were found using the `switchboardPCx` version of `find-extremes.py`. Some timestamps of extreme points are in `isg/speech/timelm/switchboardPCx/audioExamples`, and audio clips for those are in `/home/users/nigel/papers/dimensions/snippets`. Words correlating with high/low dimension values are in `switchboardPCx/countFiles/sratios`.

Version 3. Starting late 2012, I reimplemented almost everything. In particular, I separated out the PCA code from the language-modeling code, introduced `.fss` files to made feature assembly parameterizable, and documented everything. I also created some 'standard' feature specifications, including minitest/minicrunch.fss, 11 features for testing the workflow; social/symmetric.fss, 96 features, used for social speech; and fulltest/slim.fss, 78 features (48 self and 30 interlocutor), as used for the narrow-pitch work. This involved two features which are now obsolete: ph (pitch height) and pr (pitch range). This was the version shared with Columbia, Naver, and Parc.

Version 4. In Fall 2014 I began to reimplement everything again, this time in Matlab. Paola Gallardo did some of the functions, as noted in the comments. The motivations were to avoid a hybrid C-Python-Matlab workflow, to simplify the codebase, to improve portability, to use more robust features. The loss was giving up an interface able to play sound and integrated with display, labeling and user controls. This version also broke the link to the aizula code for realtime input and output, using microphone and speakers.

Version 4.1, May 2015, was the first public release, uploaded to `http://www.cs.utep.edu/nigel/midlevel/` . This version includes better extremes-finding code, more analysis tools, and handlng for multimodal features, namely gaze and game-action keystroke features.

Version 5, December 2015, was released on Github. I rewrote the documentation to stress that the code not only does PCA on prosody, but also computes a number of useful midlevel prosodic features, and to be generally clearer. The code was modified to work on Windows as well as Linux. This is the version described by this document.

10   Future Work

Implement features that relate better to human perception. The current features are inspired by perception, but for the implementation the major considerations were instead simplicity and robustness.

Clean up the codebase. The current code includes all sorts of things found useful for one project or another over the past few years.

Use a pitch tracker that also outputs probability of voicing and exploit that information.

Create an integrated workflow so that the audio at the extremes points could be easily browsed, without having to manually enter timepoints for didi or manually scroll the Elan timeline. Perhaps the matlab sound command would help.

Improve efficiency. In particular, work is repeated across features that share computations (such as narrow pitch and wide pitch), and across different window sizes of the same feature, and for same-feature-same-window-size features across different offsets, and (if the same files are being used to compute the rotation and to be rotated) for `findDimensions` and `applynormrot.m`. But for now, modularity is more valuable than efficiency.

Add more mid-level features, as hinted in Section 4.3. For example, this might include `mrate` (namely speaking rate, although in [10] we found it worse than amplitude variation (ampvar, sometimes also called jitter) as a speaking-rate proxy).

11 Local Notes

Unless otherwise specified, everything is locally in linux-side directory `/home/research/isg/speech/ppca/`.

- This file is `mlv5.tex` in `doc/`.
- The source code is in `src4/`.
- `readau.m`, `readwav.m` and `fxrapt.m` (the pitch tracker) are in `voicebox/`

One version of Didi is `/home/research/isg/speech/workingDede/dede`. If dede crashes, copy `/home/research/isg/speech/workingDede/piau-au-file.PCM` to `/tmp` and restart it.

Linux machine Lisa has 32 GB, which has been adequate for everything tried so far, but things actually run faster on an Wintel 8GB laptop.

Matlab r2014a currently runs only on the 64-bit machines, e.g. `lisa`, so be sure to login or `ssh` there, or else use r2013a, in `/opt/local/Matlab/`.

The Mid-Level Features draft document is in `/home/users/nigel/papers/learners/features.tex`

References

[1] N. G. Ward, "Automatic discovery of simply-composable prosodic elements," in *Speech Prosody*, 2014, pp. 915–919.

[2] N. G. Ward and A. Vega, "A bottom-up exploration of the dimensions of dialog state in spoken interaction," in *13th Annual SIGdial Meeting on Discourse and Dialogue*, 2012.

[3] ——, "Towards empirical dialog-state modeling and its use in language modeling," in *Interspeech*, 2012.

[4] N. G. Ward, S. D. Werner, F. Garcia, and E. Sanchis, "A prosody-based vector-space model of dialog activity for information retrieval," *Speech Communication*, vol. 68, pp. 86–96, 2015.

[5] N. G. Ward and K. A. Richart-Ruiz, "Patterns of importance variation in spoken dialog," in *14th SigDial*, 2013.

[6] N. G. Ward, C. N. Jurado, R. A. Garcia, and F. A. Ramos, "On the possibility of predicting gaze aversion to improve video-chat efficiency," in *ACM Symposium on Eye Tracking Research and Applications*, 2016.

[7] N. G. Ward, D. G. Novick, and A. Vega, "Where in dialog space does uh-huh occur?" in *Interdisciplinary Workshop on Feedback Behaviors in Dialog, at Interspeech 2012*, 2012.

[8] N. G. Ward and P. Gallardo, "Non-native differences in prosodic construction use," 2015, submitted.

[9] N. G. Ward, "Mid-level prosodic features for systematically investigating dialog prosody," 2014, manuscript.

[10] N. G. Ward, A. Vega, and T. Baumann, "Prosodic and temporal features for language modeling for dialog," *Speech Communication*, vol. 54, pp. 161–174, 2011.