Midlevel Prosodic Features Toolkit

incorporating the Prosody Principal Components Analysis Workflow

Version 7.3

Nigel Ward, University of Texas at El Paso

August 4, 2025

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 novel set of prosodic features that are robust, everywhere-computable, fairly comprehensive, reasonably fast, and proxy for several perceptual qualities. In this it has some advantages over more commonly used packages such as OpenSmile, Covarep, Voicebox, Praat, and Novoic's Surfboard. It also contains a workflow for analysis including Principal Components Analysis (PCA) and various support for automated and human-in-the-loop analyses.

This document assumes a basic familiarity with prosody and its acoustic measures, as obtainable, for example, from [?], Chapter 4 of [?] or Chapter 4 of [?].

This document is a work in progress. Comments and suggestions are welcome.

2 Getting Started

The code is at `https://github.com/nigelgward/midlevel` . It requires Matlab. (This was developed mostly on release r2015a, but seems to work on some earlier releases and forward through r2019a. r2020 apparently renamed the function `hanning` to `hann` and moved it to the Signal Processing Toolkit, so if that causes problems, it may be enough to just edit `fxrapt.m` to rename `hanning` to `hann`.)

For an initial test of whether it runs for you, download the code, enter Matlab, do addpath for midlevel/src, midlevel/src/voicebox, and midlevel/flowtest, then cd to the flowtest subdirectory, and do findDimensions('.', 'minicrunch.fss');. In a minute or so this should create a hrloadings.txt file; if so, the code is probably working fine.

If you are interested only in using the features, you can start by experimenting with computeFeatures(), then skip ahead to read Section 8, and then back to Section 5 for the file formats.

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 an inventory of audio files to process, and 3) select a feature set to use, as described in Section 5.

3 Using it as a Black Box, for Pattern Discovery

If your aim is simply to derive insight on the prosodic constructions of a language or dataset:

- download this toolkit to a directory called midlevel somewhere
- start up Matlab
- addpath for midlevel/src and midlevel/src/voicebox, with exact directory names depending on your download destination
- cd to the directory containing your audio files
- c2c();
- examine the loadings, as seen in the figures in the loadingplots directory
- listen to timepoints in your data where a dimension has extreme high or low values, as listed in the files in the extremes directory. You can do this with Elan or a similar tool, or follow the steps in src/extremeClips.awk to create clips for faster listening.

4 PCA-Based Analysis Overview

This toolkit was designed to apply Principal Components Analysis over prosodic feature collections computed over dialog datasets of an hour or so. This is useful, we have found, for various purposes. It gives dimensions which correspond to interpretable patterns of behavior [**?**]. The values of these dimensions usefully characterize the instantaneous state of the dialog [**?**]. Applications so far include language modeling, information retrieval, filtering, gaze prediction, finding patterns of action-coordinating prosody, distributional analysis, predicting actions from prosody, and examining non-native dialog patterns [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. Before reading further you may want to read 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.

4.1 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

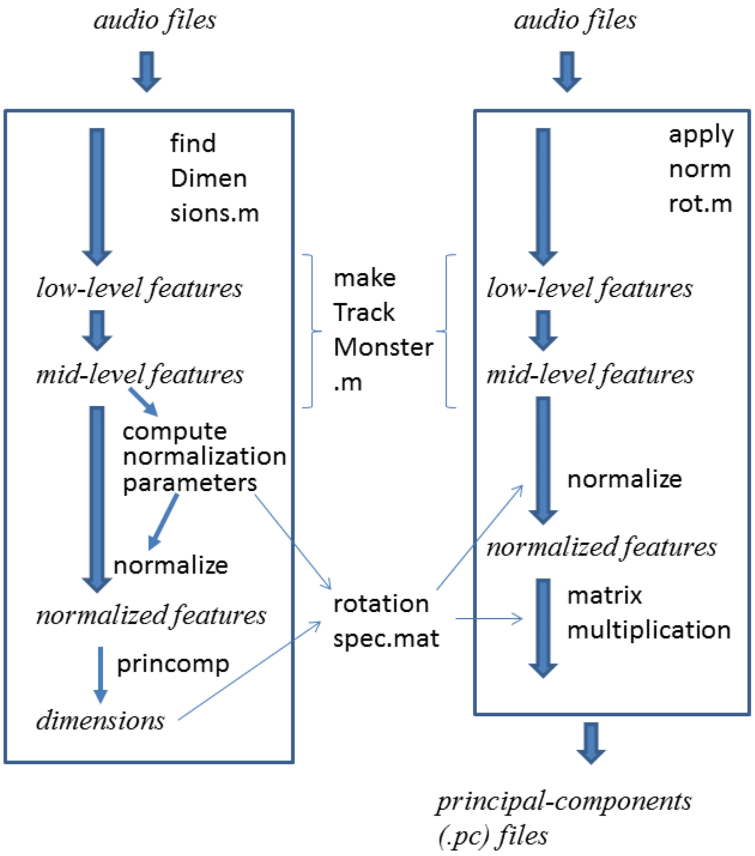

Figure 1: Workflow Overview

more sense if the file to be processed is from the same set as the audio used to generate the normalization parameters (Section 4.2), to avoid potential problems due to different domains, speaking styles, or languages. Recording conditions may also be an issue, although the features are designed to be fairly robust to these.)

Thus the function `applynormrot.m` creates a `.pc` file for each track of one or more audio files; it does this in five steps:

- 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, and also locations of dimension extremes.

The resulting dimension-based representation can then be as input to for various tasks, or can be interpreted, as described in Section 6.

4.2 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 it performs 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 4.1)

The PCA itself is done using Matlab's `princomp` function. This is memory-intensive. If you're processing a lot of audio (more than, say, 100 minutes), then first compute the rotation using just a subset, then apply it to everything. If this is not done, Matlab will gobble memory and freeze or crash the machine.

4.3 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 5.2)
featurespec file specifies the set of features to use (Section 5.3)
output dir specifies where to write the resulting `.pc` files (one per track), and extremes files (one per dimension)

and two are locations which are set implicitly:

pitch cache is the subdirectory where to store (or find) the f0 values as computed by `fxrapt`, 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.

5 File Formats

5.1 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, 16 bit, linear PCM, 8K sampling, of no more than 10 minutes. While sometimes longer files, other `.au` formats, and even `.wav` files, often work fine, it is safer to use this format. The `sox` program is convenient for conversions to this format. The workflow was originally designed to handle stereo recordings, with each speaker in a separate track, but monoaural data can also be handled. If an mono `.fss` file (see below) is used.

—**f0.mat** a file specifying for an audio track the pitch every 10 milliseconds. These files are created because `fxrapt` is time-consuming, so it's worth caching the results to avoid recomputing 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 usually commented out.

5.2 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`.

5.3 Feature Specification Files

To encode contextual information we need to use features computed at various temporal offsets, relative to the point of interest.

As a default this point of interest is stepped through the audio, every 10 milliseconds. 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.

Many `.fss` files exist, each designed for a specific purpose. `april.fss`, has more features for the primary-track talker than for the other talker, so can be used to analyze one speaker's behavior. `medslim.fss` has features evenly distributed across the two speakers. `mono4.fss` is useful for information retrieval in broadcast news. `minicrunch.fss` is a small set for testing the workflow

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 intensity (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, “inte” to the other track, containing the interlocutor’s voice. The primary track is specified in the tracklist file, as either left (`l`) or right (`r`). If a fss file has only self features (no “inte” features) it can be applied to mono files.

In fss files currently the following codes are recognized:

vo	intensity (volume)
vf	voicing fraction
sf	speaking fraction
sr	speaking-rate proxy (also reflects enunciated vs. reduced, and creaky vs. modal)
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

pd	late (delayed) pitch peak
le	lengthening, of a vowel etc.
en	enunciation (articulatory precision) via vowel distinctiveness
re	phonetic reduction via vowel centralization
hr	HuBert-based reduction estimates; seelookupReductionEst.m
vr	voiced-unvoiced intensity ratio

ts	time since start of the recording (window start/end times ignored)
te	time until end of the recording (ditto)
ns	nearness to start of the recording (ditto)
ne	nearness to end of the recording (ditto)

cp	smoothed cepstral peak prominence, representing breathiness, etc.
st	spectral tilt, higher (negative but non-zero) values, maybe emphasis

Reserved two-letter code (implementation pending):

ha	harmonicity
br	breathiness

Adding a new prosodic feature requires changing a few things. First you create an entry for your new feature in the featurespec file, choosing any unused 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` statement in `makeTrackMonster` to associate your new feature-computing function with the two-letter code. Fourth, you add it to the list in `getfeaturespec`.

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.

All feature-computing functions must return values everywhere, even at the start and end of the audio file. In particular, while raw pitch features can include NaNs, the mid-level features must be designed not to. Mid-level features with windows longer than 10ms, when computed at timepoints 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 fully meaningful value. In such cases the function should return an non-obtrusive value in the typical range (rather than some extreme value like -9999, since that would mess up the normalization). While such values can be problematic, this is not usually a problem, since the audio files we work with are usually long enough (generally 2–10 minutes long) that the vast majority of features values will be valid.

5.4 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.

6 Interpreting the Dimensions

An important reason to use Principal Components Analysis is to understand what's going on in the data. For all data sets tried so far, the dimensions output by PCA have turned out to be readily interpretable as meaningful patterns of behavior.

One preliminary is to look at the variance and cumulative variance explained by the PCA-found dimensions, which is written to `variance.txt` by findDimensions.

Then there are three techniques to help you arrive at interpretations:

6.1 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. This second parameter is optional unless you're using an rotationspec that was written by a very early version of `findDimensions`.

This creates a `loadingplots` directory and writes a diagram per dimension as a `.png` file.

6.2 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, the files `dim00.txt` etc. are written in the `extremes` subdirectory of outdir by `findExtremes.m` (called by `applynormrot.m`, as described in Section 4.1). 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, and then 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. Elan and VLC are also good choices.

6.3 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. However this is only possible if we have transcribed data, and even then, since the file formats vary, it requires custom code for each case.

7 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 basic statistics in how the distributions on the dimensions differ. (While helpful, this certainly does not give the whole picture [?].)

These can be seen in `summary-stats.txt`, which is written by `applynormrot.m`. Useful may be the average value (to detect bias to one side of the dimensions), the standard deviation (to detect failure to use a dimension much), etc.

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

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

8 Features

8.1 Frame-Level Feature Computations

Three frame-level (low-level) features are computed: pitch, energy, and cepstrum.

The raw log-energy is simply

$$e^r_f = log\sqrt{(\frac{1}{160}\sum_{i=f-80}^{f+79} s_i^2)} \tag{1}$$

where f is the frame center and s is the signal, assuming the sampling rate is 16000 per second and the frames are 10 milliseconds long. This is computed in `computeLogEnergy.m`.

The raw pitch,

$$p_f^r \tag{2}$$

is obtained with `lookupOrComputePitch.m`, which is a wrapper for various functions as described in the next section. This returns values in hertz, or NaNs if there is no detectable pitch.

The cepstrum is computed using `mfcc.m`.

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

Frame-level features will include multimodal features, if so specified in the `.fss` file. If keystrokes are specified, `featurizeKeystrokes.m` is called to load that information; similarly `featurizeGaze.m` is called if gaze features are specified.

8.2 Frame-Level Feature Normalization

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

$$p_f^p = percentile(p_f^r) \tag{3}$$

where the percentile is based on the distribution of pitch values in all voiced regions in the entire track. These are mapped to the range 0 to 1; thus the lowest pitch value in the track maps to 0.0, the highest to 1.0, and the median pitch value to 0.5. As described below, percentilized pitch is used for the highness and lowless features; but raw pitch is used for the for creaky, wide and narrow features.

Energy is normalized as described below. (It is not done at the frame level, but over larger windows, because frame-level energy may be less clearly bimodal.)

8.3 Mid-Level Feature Computation

The mid-level features are as listed in Section 5.3. Each summarizes something about the values of the frame-level features across some window. The motivations for choosing these specific features and these specific implementations are given in Chapter 9 of [**?**]. There are many desiderata for feature sets, some in conflict. For this feature set, the primary considerations were that it be everywhere-defined, robust, and simple. Other considerations were that they be useful, have at least some perceptual validity, and are reasonably fast to compute. All of these were explored spottily, not systematically. Compatibility with previous definitions or implementations was not a priority.

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. Typically these are downsampled before use in applications.

These are intended to extract essentially all the useful information in the frame-level features; for machine-learning applications, the mid-level features should be the ones to use.

8.3.1 Window Energy

We scale the energy to normalize for individual differences and recording-condition differences, specifically regarding typical speaking volume and noise floor. To do this we find typical-silence and typical-speech values of energy, using `findClusterMeans.m`. This implements a simplified 1-dimensional k-means algorithm (k=2) and applies it to over all e^r values in the track, using as seeds the min and max of e^r_f over the entire track. We then normalize the energy with respect to these values.

$$e_f = \frac{e^r_f - e^{silence}}{e^{speech} - e^{silence}} \tag{4}$$

where $e^{silence}$ is the typical silence energy and e^{speech} is typical speech energy. Thus the resulting e is on a scale where typical vowel volumes are 1 and typical silent frames are 0.

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.

8.3.2 Pitch Highness

$$h_{a,b} = \frac{1}{b-a} \sum_{i=a}^{b} (p^p_i - .5 | p^p_i > .5 \, \text{and} \, p_i \neq NaN) \tag{5}$$

where a is the start of the window and b is the end, both in frames. We use windows of various sizes, thus $a - b$ may be 5 (for a 50 milliseconds window), 10 (for a 10 ms window), etc.

Note that this computation uses the pitch percentile values.

8.3.3 Pitch Lowness

$$l_{a,b} = \frac{1}{b-a} \sum_{i=a}^{b} (.5 - p^p_i | p^p_i < .5 \, \text{and} \, p_i \neq NaN) \tag{6}$$

8.3.4 Creakiness

$$
\begin{aligned}
c_{a,b} = \frac{1}{b-a}\sum_{i=a}^{b} 1 \text{ if } .475 < \frac{p_i}{p_{i+1}} < .525 \text{ or} \\
.80 < \frac{p_i}{p_{i+1}} < .95 \text{ or} \\
1.05 < \frac{p_i}{p_{i+1}} < 1.25 \text{ or} \\
1.90 < \frac{p_i}{p_{i+1}} < 2.10 \quad (7)
\end{aligned}
$$

Creak can be seen to affect computed F_0 in two main ways: the presence of octave jumps and the presence of frame-to-frame pitch jumps beyond what one would normally expect [?].

The thresholds are based roughly on what is known about maximum human-achievable variation in pitch. This is reported to be 61.3 semitones per second up and 70.6 semitones per second down[?]. Since a semitone is a 6% rise or fall, this means excursions greater than 3.6% per frame up or 4.2% per frame down are not normal pitch movements.

Accordingly, in the equation the first clause detects pitch halving: a pitch point within 5% of half the value of a neighboring pitch point counts as evidence for creaky voice. Similarly the last clause considers a pitch point that is within 5% of twice the value of a neighboring pitch point to also be evidence for doubling and thus creaky voice. The tolerance (5%) is a little wider than Xu's results would imply, mostly because pitch tracking of spontaneous speech is not always highly accurate.

The second and third clauses detect variation in pitch that is too large to be a normal pitch movement. Again the 5% criterion is used, for the same reason.

To explain this equation in another way, every pair of pitch points counts as evidence for creaky voice except, a) those that are probably due to normal pitch movements, with a ratio of between 0.95 and 1.05, which are the vast majority, and b) those that are probably due to noise of some kind, namely those with a ratio below .475 or above 2.10, or a ratio between 0.525 and 0.80, or a ratio between 1.25 and 1.90. These specific thresholds are based purely on experience.

8.3.5 Narrowness

$$
n_{a,b} = \frac{1}{b-a}\sum_{i=a}^{b}\sum_{i-50}^{i+49} 1 \quad \text{if } 0.98 < \frac{p_i}{p_{i-1}} < 1.02 \qquad (8)
$$

where 50 refers to 50 frames. With the nested *for* loop this is somewhat inefficient, but this has not been a problem in practice.

This computation sums up the evidence for some narrow (flat) pitch involving the window from a to b. While there are many ways in which this could be done, this method was chosen in part because it is robust to pitch halving and doubling, but mostly because it is simple.

8.3.6 Wideness

$$w_{a,b} = \frac{1}{b-a} \sum_{i=a}^{b} \sum_{i-50}^{i+49} 1 \;\; \text{if}\, 0.7 < \frac{p_i}{p_{i-1}} < 0.9 \,\text{or}\, 1.1 < \frac{p_i}{p_{i-1}} < 1.3 \tag{9}$$

Note that if the ratio is more extreme, one of the two pitch points is likely to be spurious, so we don't consider such cases to be evidence.

8.3.7 Window Energy:

$$e^w = \frac{1}{b-a} \sum_{i=a}^{b} e_i \tag{10}$$

8.3.8 Energy Flux

$$r = \frac{1}{b-a} \sum_{i=a}^{b-1} |e_i - e_{i+1}| \tag{11}$$

This was our first attempt at a proxy for rate. It seems to correlate also with the carefulness of articulation, and with creaky voice.

**** `mrate` (namely speaking rate, although in [**?**] we found it worse than amplitude variation (ampvar, sometimes also called jitter) as a speaking-rate proxy).

8.3.9 Late Pitch Peak

This is described in the comments in epeakness.m, ppeakness.m, computeSlip.m, computeWindowedSlips.m, and laplacianOfGaussian.m.

8.3.10 Lengthening

This is described in lengthening.m, and also in doc/lengthening-rate.txt.

8.3.11 Enunciation and Reduction

One measure of phonetic reduction vs enuciation is the degree to which vowels are centralized vs distinct This function measures, as a proxy for that proxy, how spectrally close or far from the average are the voiced segments, returns how often/strongly this happens as the degree of evidence for reduction or enunciation, respectively. It's not clear what this really measures, given especially that it doesn't relate in any simple way to the output of the new hr (Hubert-based reduction) feature.

8.3.12 Voiced-Unvoiced Intensity Ratio

This is described in the comments of voicedUnvoicedIR.m.

8.3.13 Temporal Features

The features `ts`, `te`, `ns`, and `nr` are computed directly in makeTrackMonster.m

8.4 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, after each is 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 in `makeTrackMonster` etc. .

8.5 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.)

9 Pitch Trackers

At the root of many of the midlevel features is the frame-level pitch. The toolkit supports two ways of computing this.

The first is Mike Brookes's Voicebox function `fxrapt.m`, which is conveniently written in Matlab and very well documented.

The second is David Talkin's Reaper. I added because it can handle the Switchboard files that `fxrapt` cannot. (I'm guessing that the problem is due to recording issues with some files, as discussed in `https://catalog.ldc.upenn.edu/docs/LDC97S62/swb1_manual.txt` in Section 13.) Another advantage is that it is about 7 times faster.

(Other alternatives explored were Matlab's built-in pitch tracker, which seems to have a lot of outliers and false alarms, and `Yappt`, which also fails on many Switchboard files, including some that `fxrapt` could handle, and `Praat`, which seems slow and awkward for integrating.)

Reaper is in C++, and it runs only on *mono* files, so the integration with the midlevel toolkit is awkward. The bash code `reaperize.sh` does what's needed for processing Switchboard files, and serves as an illustration of the necessary steps.

Since the midlevel features depend on the pitch detector, and were tuned (somewhat) to work well with the quirks of `fxrapt`, it's worth noting that, on a tiny sample, `reaper`'s output looked visually smoother and more accurate than that of `fxrapt`, which likely changes the behavior of the creaky voice feature.

Another issue is the question of whether to set the bounds differently for male and female speakers. This makes a huge difference for `Praat`, but appears not to matter much at all for `reaper`, so I don't.

10 Robustness

The toolkit is designed to be robust to normal human voice variation and modest noise, but it is not robust to every form of corruption. Files that contain splices mess up the default pitch tracker, as noted above, and also the cepstrum computation. The symptom in both cases manifests as NaN values. The easiest cure is to exclude the offending file, e.g. English Callhome 4065, or to trim the file to exclude the troublesome region, for example with `sox bad.au cleaned.au fade 0 -1 0.01`, as for some Switchboard files. Background music can also be a problem.

11 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. A small test harness is included as `validateFeature.m`.

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`.)

12 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 and made it work for features at different offsets and over different window sizes.

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.

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 gains were avoiding a hybrid C-Python-Matlab workflow, simplifying the codebase, improving portability, and improved robustness. 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 handling 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 made to work on Windows as well as Linux.

Version 6, April 2016. I added the late-pitch-peak feature and documented the feature computations. This work was done at Kyoto University. While there Divesh Lala and Narimasa Watanabe kindly provided helpful comments on the code and documentation.

Version 7, April 2017. I added the cepstrum-based features (`mfcc.m` and related files), and the temporal features. Thanks to Kamil Wojcicki for writing this code.

Version 7.1, May 2019. With the kind permission of Mike Brookes, I copied all needed functions from Voicebox (for audio file input and the pitch computation) into the new `voicebox` subdirectory, so a separate download is no longer needed.

Version 7.2, February 2020. I added an interface to alternatively use `reaper`-derived pitch values, and simplified the getting-started procedure to as a default process all `au` or `wav` files in the current directory, thus avoiding the need to create a tracklist file.

Version 7.3, November 2020. Marcin Wlodarczak's implementation of Smooted Cepstral Peak Prominence added. Note: This is very time consuming; taking as much time as a pitch computation.

13 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 can be easily browsed, without having to manually enter timepoints for didi or manually scroll the Elan timeline. Perhaps use the matlab `sound` command for this.

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.

Systematize the way padding is done.

14 Location Notes

The repository is https://github.com/nigelgward/midlevel/ .

This file is in the doc directory, as mlv7.tex and mlv7.pdf. The code is in the src directory.

References

[1] N. G. Ward, *Prosodic Patterns in English Conversation.* Cambridge University Press, 2019.

[2] R. Ogden, *An Introduction to English Phonetics, 2nd Edition.* Edinburgh University Press, 2017.

[3] P. Ladefoged, *Phonetic Data Analysis.* Blackwell, 2003.

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

[5] 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.

[6] ——, “Towards empirical dialog-state modeling and its use in language modeling,” in *Interspeech*, 2012.

[7] 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.

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

[9] 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.

[10] N. G. Ward and S. Abu, “Action-coordinating prosody,” in *Speech Prosody*, 2016.

[11] 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.

[12] N. G. Ward and P. Gallardo, “Non-native differences in prosodic construction use,” *Dialogue and Discourse*, vol. 8, pp. 1–31, 2017.

[13] N. G. Ward, Y. Li, T. Zhao, and T. Kawahara, “Interactional and pragmatics-related prosodic patterns in Mandarin dialog,” in *Speech Prosody*, 2016.

[14] N. G. Ward, J. C. Carlson, O. Fuentes, D. Castan, E. Shriberg, and A. Tsiartas, “Inferring stance from prosody,” in *Interspeech*, 2017.

[15] P. Keating, M. Garellek, and J. Kreiman, “Acoustic properties of different kinds of creaky voice,” in *International Congress of the Phonetic Sciences*, 2015.

[16] Y. Xu and X. Sun, “Maximum speed of pitch change and how it may relate to speech,” *The Journal of the Acoustical Society of America*, vol. 111, pp. 1399–1413, 2002.

[17] 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.