Prosody Principal Components Analysis (PPCA) Version 4 (Matlab Version) Workflow Notes

Nigel Ward, University of Texas at El Paso

February 18, 2015

Abstract: Applying Principal Component Analysis to prosodic features has been useful for several tasks. This report describes the Matlab-based workflow and code.

1 Background

Principal Components Analysis applied to a large set of prosodic features spanning various temporal windows can be useful in various ways. This gives dimensions which correspond to interpretable patterns [Ward, 2014]. The values of these dimensions usefully character the instantaneous state of the dialog [Ward and Vega, 2012a]. Applications of these include language modeling, information retrieval, filtering, and linguistic inquiry [Ward and Vega, 2012b, Ward et al., 2015, Ward and Richart-Ruiz, 2013, Ward et al., 2012]. Ongoing work includes: 1) use of the dimensions for speech synthesis, 2) examining patterns of learner behavior in terms of the dimensions. We also foresee applying these techniques to examining differences among individuals in their use of the dimensions.

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

2 Overview

There are two main use cases. Figure 1 overviews how they relate.

2.1 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

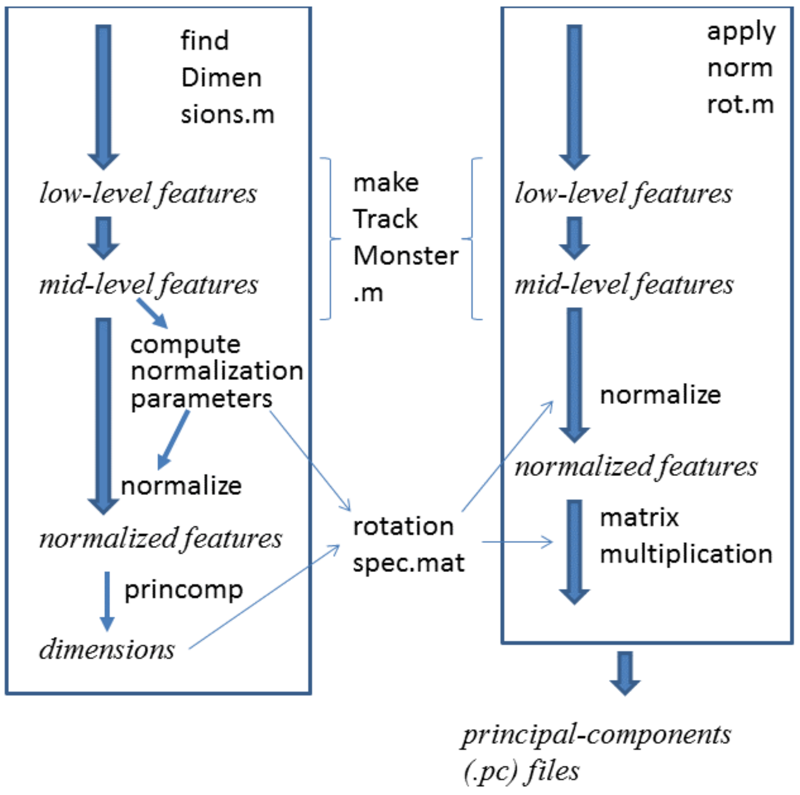

Figure 1: Workflow Overview

more sense if the file to be processed is from the same set as the audio used to generate the normalization parameters (Section 2.2), 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
- compute the raw features
- normalize them, using some precomputed parameters (means and standard deviations)
- rotate them, using a precomputed rotation
- write the results to `.pc` files.

The resulting dimensional representation can then be as input to machine-learning algorithms for various tasks, or can be interpreted.

2.2 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

- 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 norm params for later use (Section 2.1)

2.3 Overview of the Arguments

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

tracklist specifies which tracks to process, each being a track from an audio file (Section 3.2)
featurespec file specifies the set of features to use (Section 3.3)
output dir specifies where to write the resulting `.pc` files (one per track)

The other two things are locations which are, implicitly, the location where the matlab process is run.

pitch cache subdirectory where to store (or find) the `fxrapt`-output f0 values, as `.mat` files
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.

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

3 File Types

3.1 Data Files

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

—.au, .wav The input. A stereo audio file. `.wav` files sometimes cause trouble for `fxrapt`, so it's better to convert them first to `.au` files. Traditionally these have been in Sun format, specifically 16 bit, linear PCM, 8K sampling.
—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 ever 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.

3.2 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`.

3.3 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 originally they described how to crunch together data from individual feature files into a single composite file suitable for machine learning or dimensionality reduction.

A current feature set under development is `fulltest/al.fss`, an "assumption light" new set of mid-level features, including about 168 features

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

```
vo -100 -200 self
cr -200 400 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. Since `getfeaturespec.m` parses these files by fixed offsets, it's important for everything to line up exactly, and to use spaces not tabs.

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
wp	wide pitch range
hp	high pitch: degree of highness
lp	low pitch: degree of highness

Reserved two-letter codes:

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

3.4 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.

4 Support for Examining and Interpreting the Results

Examination of intermediate and final results is important, both to check that everything is working properly, and to interpret the results of the process.

4.1 Examining the Features

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.

4.2 Examining the Correlations

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 two correlation files: `pre-norm-corr.txt` and `post-norm-corr.txt`, each showing the most highly correlated and most anticorrelated features for each othe feature. These are output by `output_correlations.m`.

4.3 Examining Statistics about the Dimensions

To see the variance and cumulative variance explained by the PCA-found dimensions, load the `rotationspec.mat` file and process it with:

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

More interestingly, 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 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

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`. There is also fragments of a workflow described in `histo/README.txt`: in short, this uses `distDist.m`, `bhatd.m`, and `binProbs.m` to generate histograms for each dimension, including superimposed histograms for the two populations, and to compute the Bhattacharyya distance.

4.4 Interpreting the Dimensions

To understand the dimensions, there are three methods to apply.

4.4.1 Examine the Factor Loadings

`findDimensions.m` includes a call to `writeloadings.m`, which writes a large, human-readable file called `loadings.txt`, the lines of which give the loadings of each feature on each dimension, 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
...
```

These files can then be examined to understand the nature of each dimension. It's particularly useful to first 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 the Unix `sort` and `grep` commands to find, for example, the features with the highest loadings and those with the strongest (highest absolute) loadings.

There is also visualization code in `ppattern.m`. **

4.4.2 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. This is done by going into the `.pc` files and finding the timepoints with extreme values, using `find-extremes.py` This 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.

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. Invokability from the command line is a big plus. Using second notation (not minutes and seconds) is also nice. Dede does these things, but only the 32-bit linux machines run it. One version is in `/home/research/isg/speech/workingDede/dede`. If dede crashes, copy `/home/research/isg/speech/workingDede/piau-au-file.PCM` to `/tmp` and restart it.

In future, it might be nice to automatically feed timepoints to `dede`, to direct it to the right places without requiring the user to view and re-specifiy timepoints.

4.4.3 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.

5 Internals

5.1 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.

5.2 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 find_ss_cluster_means.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 tracks 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. (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.)

5.3 Mid-Level Feature Computation

The mid-level features are as listed in Section 3.3. Each summarizes something about the values of the frame-level features across some window. The motivations for these specific choices of feature are in another document: Mid-Level Prosodic Features for Systematically Investigating Dialog Prosody (in preparation).

Each value is associated with the time at the center of the window. Windows are shifted (stepped) every 20ms, because it's unlikely that prosodic features change faster than that. Windows are always at least 50ms long, thus they are overlapped.

5.4 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.

5.5 Overall Normalization

Before doing PCA we need to normalize the features to all have zero mean. It's also helpful for them to have approximately the same standard deviations, to avoid features with larger variance dominating. (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, and we don't want to lose that information[1] (although some has already been removed by the time we get here, by the normalization of the pitch and the energy). Thus we normalize over the monster array, that is, over the data in all the files in some large set; the same large set of data that we'll use for the PCA.

5.6 Determining the Rotation (doing the PCA)

The PCA itself is done using Matlab's `princomp` function. This is memory-intensive.

5.7 Rotating

As noted in Section 2.1, this is done by `applynormrot.m`, which applies a previously saved `rotationspec.mat`, namely the one found in the current directory.

While this and the previous step could be packaged together, currently they are separate. (Packaging them together would be convenient for those times when the files used to determine the rotation are the same as those we wish to rotate. Doing so would in addition speed things up, by avoiding the need to once again read in all the audio files, compute the features, and normalize them. However the two steps are separate for now because we also sometimes need to determine the dimensions based on one set of data, then apply it to a new set of data.)

[1]While it's fine to normalize over an entire set of dialogs to bring the overall mean to zero, when Shreyas tried normalizing, file-by-file, to have each individual file have zero mean, all language-modeling benefit was lost.

6 Validation

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

7 History

Version 1. In our language modeling 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 factors 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. There are several classic old 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 not 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 big downside is that for labeling and analysis, Matlab doesn't seem to support sound integrated with a display, labeling and user controls, so for those aspects of the work we still use Elan and dede. In this version we've also broken the link to the aizula code for realtime input and output, using microphone and speakers.

8 Future Work

It would be nice to use a pitch tracker that also outputs probability of voicing.

The implementation could be made much more efficient. 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.

Other mid-level features could be added, as hinted in Section 3.3. For example, this might include `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).

... Find_extremes.py should be redone in Matlab. Because adjacent timepoints typically have similar values, existing extremes files contain lots of timepoints which are close together. It might be nice to add a pruning stage where we drop all points within 1000ms of a more extreme point. Also it might be interesting to be able to find and peruse 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. One way to do this might be: 1. In matlab, find the norm (the mean absolute value) for every feature. 2. Divide all feature values by their feature's norm. 3. 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. 4. output the filename, the timepoint, the normalized value for the feature of interest, the salience, and the name of the strongest rival feature.]

9 Local Notes

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

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

Linux machine Lisa has 32 GB, which has been adequate for everything tried so far.

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 document is in `/home/users/nigel/paers/learners/features.tex`

References

[Ward, 2014] Ward, N. G. (2014). Automatic discovery of simply-composable prosodic elements. In *Speech Prosody*, pages 915–919.

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

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

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

[Ward and Vega, 2012b] Ward, N. G. and Vega, A. (2012b). Towards empirical dialog-state modeling and its use in language modeling. In *Interspeech*.

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