Report on Implementation of Unsupervised Pattern Discovery in Speech

Gu Yu, Lyu Kefan

June 17, 2017

1 Overview

In this project, we implemented the algorithms in [1], which discovers the recurring patterns in speech. Our implementation does not agree with the original in every detail, but follows roughly the same framework. Simplifications are made sometimes, to simplify programming, or to bypass certain unclearness in the original paper.

We programmed the algorithms in Python 3, with the assistance of external libraries, NumPy (for array computation) and librosa (for signal manipulation).

Our implementation is available on GitHub [3], which also includes some test data and results.

2 Components

2.1 Preparing Data

Given an audio, we first divide it into short fragments by simple silence detection. We remove all fragments whose duration is shorter than 0.5s. We use the whitened Mel-Frequency Cepstral Coefficients (MFCC) of each fragment to represent it. We could not find a precise definition of “whitening” referred to here, but we thought that PCA related meaning which is popular in the context of machine learning is not likely applicable. Finally we used the definition in another work [2], but this definition cannot be that the authors of [1] used.

After the transformation, we get a sequence of short audio signals. For each pair of short fragments, we perform matching with Segmental Dynamic Time Warping (SDTW) as described below.

2.2 Segmental DTW

For a pair of MFCC sequences, we use SDTW to detect the similar subsequences. As described in [1], we first compute the cost of matching two features, which is defined to be their Euclidean distance. Next, we compute an optimal match path for each band-constrained diagonal region of the cost table, with the unchanged classic dynamic programming technique. After this stage, we have a number of match paths, each of which aligns with the onset of one of the two signals.

The next step is to extract from these paths the indeed similar subsequences. Length-constrained minimum average (LCMA) algorithm is introduced for this task. The algorithm finds in each path a subsequence that minimizes its mean cost. The resulting path have length ranging from L (a specified value) to $2L-1$. After this stage, the minimum average value, referred to as average distortion, will be used to decide whether to discard the path or not.

In the original paper, the threshold for distortion is 2.5. But in our implementation, features are normalized thus their difference has norm at most 2. With the same consideration, we set our threshold to be 0.95. Paths with average distortion above this value will be discarded.

These path segments are passed on to the next stage of analysis.

2.3 Building the Adjacency Graph

The most straightforward way to construct a graph would be to transform every sample in time (50 per second) into a node, but that will result in a too large graph, therefore we seek means to extract a small portion of these sample points that are representative of its neighboring points.

[1] defines a similarity score for a pair of points, who share a common path, which is negatively correlated with average distortion of that path, and then count the scores for each point. Scores will fluctuate over time, peaking at phonemes that recur often, and touching zero at boundaries of utterances. The scores are smoothed, after which local extrema are transformed into nodes of the adjacency graph.

Each path will then be transformed into edges, connecting any two nodes that both appear in the duration (two disjoint intervals along t-axis) of the path. The weight on that edge is simply the similarity score of the path.

The final part of the analysis is to find the clusters in this graph.

2.4 Finding Clusters

We try to find clusters in the graph remain. A good clustering means nodes in a group are more densely connected to each other than they are to the nodes outside. Here we use a modularity measure Q and a greedy algorithm to find a good clustering (not optimal)

$$Q = \sum_i (e_{ii} - a_i^2)$$

where e_{ij} is the fraction of edges in the origin network that connect goup i and group j. $a_i = \sum_j e_{ij}$. So Q is the fraction of edges that fall within groups minus the expected value when it is random distributed. It cost much to find the optimal value of Q. So we use a greedy algorithm.

At first, each node makes a single group. Each time we combine two groups x, y that:

$$(x, y) = \arg\max Q(G - x - y + x \wedge y)$$

$$\Delta Q = e_{x\wedge y} - a_{x\wedge y}^2 - (e_{xx} - a_x^2) - (e_{yy} - a_y^2) = 2(e_{xy} - a_x a_y)$$

So

$$(x, y) = \arg\max(e_{xy} - a_x a_y)$$

So each time we find such two groups and combine them. Refresh the data and judge stop or not.

Notice that the sign of $\max(e_{xy} - a_x a_y)$ will change at most one time during the algorithm. So $Q(k)$ has only one peek. We prefer mistakenly leaving two like groups unmerged than joining two unlike groups. So we choose the stop point at 80% of peak modularity.

3 Testing

[1] tested the algorithm on several hour-long lecture records, and discovered some of the recurring English words (or phrases). However, we did not test our implementation on these audios. What we used for testing, is an audio record of Gu Yu reading the first a few hundred digits of π. Gu Yu reads the digits in Chinese, mostly pausing between every four digits. The audio lasts 1min.

The reason we have chosen this dataset is threefold. First, our implementation in Python cannot run very fast, nor are our laptops powerful computers, we cannot analyze long speeches within limited time. Second, even if our program is able to analyze a long speech, we cannot efficiently validate the results without existing benchmarks at hand. However, our π-digits dataset has a relatively small vocabulary (11 words, 0-9 plus '.', dian3), and words are all monosyllabic. This largely accelerates programming and debugging, and produces approachable outputs that make it easy to validate our implementation. Finally, even though the task is inarguably easier than the task proposed in the original paper, it is not *substantially* easier. Many of challenges that are present in the original task still remain. For instance, the distortion between utterances of the same words, and liaison, etc. Therefore we still regard the task as proper for our testing purpose.

We realized that many of the parameters involved are not task-independent. On completion of our first commit, we have hoped that it will directly work on a 5min clip of a lecture. Later, we replace the task with π-digits task, but without much improvement. It was not until a few rounds of debugging the main logic has been done in vain, that we realized the parameters are decisive on the behavior.

The parameters we made adjustable are R, L, θ, smoothing window width and number of MFCCs, as defined in `config.py`. We start by tuning the number of MFCCs, which we did not well understand and was mostly used as a black box. The choosing of L is finally determined s.t. it covers the estimated length of the utterance of a single digit (0.28s). Then we tried different combinations

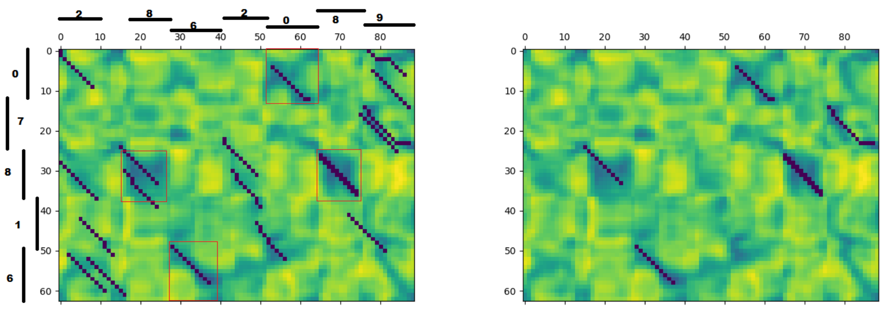

Figure 1: Before and after properly setting the distortion threshold. Red boxes for correct matches and Dark blue pixels for matching paths.

of R and θ to optimize the performance on a few particular pairs. Figure below indicates the performance on a particular pair before and after tuning.

The final result of clustering is shown in the table below:

Group ID	occurrences	transcription
0	11	3
1	4	1
2	2	1, 14
4	20	6 or 9
5	9	2
8	15	8
9	9	7
14	7	4
25	1	5
26	1	50
42	1	0
47	1	5
54	1	06
56	1	6
57	1	02

Table 1: Cluster found in π-digits

We noticed that modularity Q does not decline until termination, which may be due to relatively small size of our graph. This causes the termination criterion 80% peak value to be inapplicable. We therefore set the number of clusters explicitly as 15, which is slightly more than desired to tolerance strange nodes.

We observe that most digits are grouped correctly, except for the case of 6 and 9 (liu4 and jiu3). They are joined together at a rather early stage of clustering, it seems that 6 and 9 are more similar to each other than one instance of 3 (san1) and another 3. We can give the explanation that 6 and 9 share the same vowel that lasts relatively longer. Besides, although 5 is not wrongly categorized, the node extraction stage produces very few number of utterance of 5 (wu3). This is probably due to the utterances of 5 have relatively low amplitudes.

4 Conclusion

We implemented the algorithm proposed in [1]. While recognizing it power, we also note several shortcomings of it. First, the parameters are task-specific, which adds cost to migrating the algorithm between datasets. Second, the setting of L limits the possible length of a phonetic unit to a narrow range. Under the current framework, patterns lasting 0.5s and 4s are not likely to be found simultaneously. (More strictly, this is possible, but needs to rely on subsequent clustering stage.) A more flexible matching algorithm during path-refinement stage may help alleviate the problem.

References

[1] Park, Alex S., and James R. Glass. "Unsupervised pattern discovery in speech." IEEE Transactions on Audio, Speech, and Language Processing 16.1 (2008): 186-197.

[2] Kanadje, Manish, et al. "Assisted keyword indexing for lecture videos using unsupervised keyword spotting." Pattern Recognition Letters 71 (2016): 8-15.

[3] GitHub link to our project: https://github.com/Crispher/Speech-Science