Report of Final Project
Linear Discriminative Analysis using Power Iteration

Heinrich Dinkel
ID: 1140339107
E-mail: heinrich.dinkel@sjtu.edu.cn

1 Introduction

My topic I did my final Project about is related to my field of research, which is Speaker Recognition. In this field of research we try to find suitable model, which can predict if a given speech segment (called utterance) was spoken by a specific speaker. Generally speaking, we have usually a given set of $\mathbf{S} = \{S_1, S_2, \ldots, S_n\}$ speakers of which each produce an arbitrary mount (q) speech segments, which we have as our basic data. We try to find the most probable hypothesis that a newly, unheard-of of speech segment, belongs to any of the given speaker set $\mathbf{S}$. To accomplish this task we assume having two different models, which try to identify a newly heard utterance u:

$$H_0 = u \text{ is spoken by } S$$
$$H_1 = u \text{ is spoken not by } S$$

H_0, H_1 represent here the models for the correct speakers (H_0) and the wrong ones (or impostors H_1). So we need only to define a threshold t which indicates if the utterance u is either belonging to H_0 or H_1. Then we can rewrite:

$$\frac{P(u|H_0)}{P(u|H_1)} = \begin{cases} \leq t, \text{ accept } H_0 \\ > t, \text{ accept } H_1 \end{cases}$$

The important part is to find robust and versatile models for H_0 and H_1 respectively.

In the field of speech recognition, the input consists of raw waveforms, which will be transformed into so called features. Features are basically an extracted vectorized form of the original signal. Their purpose is to reduce the dimensionality complexity for further processing, since the overall amount of raw data is simply too arduous to handle.

2 Motivation

3 Linear Discriminative Analysis

For speaker recognition we use LDA to improve our improve our scores. It is widely used after the computation of the models H_0 and H_1 to project the resulting high-dimensional vectors into a lower dimensional space, where the separation between the involving classes is maximized.

To achieve a dimensionality reduction, we use

Assume having c classes in an arbitrary space. We calculate the proportion in between the

between-class covariance and the within-class covariance, having the current feature vector $\mathbf{x}_i$:

$$\mathcal{L}(\mathbf{w}) = -\frac{\mathbf{w}^T \Sigma_{\text{between}} \mathbf{w}}{\mathbf{w}^T \Sigma_{\text{within}} \mathbf{w}}$$

$$\Sigma_{\text{within}} = \sum_{i}^{c} \Sigma_i$$

$$\Sigma_i = \sum_{i}^{c} (\mathbf{x} - \boldsymbol{\mu}_i)(\mathbf{x} - \boldsymbol{\mu}_i)^T$$

$$\Sigma_{\text{between}} = N_i \sum_{i}^{c} (\boldsymbol{\mu}_i - \boldsymbol{\mu})(\boldsymbol{\mu}_i - \boldsymbol{\mu})^T$$

where $\boldsymbol{\mu}_i$ is the mean of the current class, $\boldsymbol{\mu}$ the overall mean and N_i is the sample mean of the i-th class.

As soon as the covariances are calculated, we need to focus on how to compute the weight vectors $\mathbf{w}$. We solve this generalized eigenvalue problem by calculating the eigenvalues of:

$$\sigma\left(\Sigma_{\text{within}}^{-1} \Sigma_{\text{between}}\right) = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$$

With these eigenvalues we compute the corresponding eigenvectors. The clue of the whole algorithm is then to use the most important $m, m < n$ eigenvalues, to construct the overall weight matrix $\boldsymbol{W}$, which then projects the input features into the new m dimensional feature space.

$$\mathbf{y} = \mathbf{W}^T \mathbf{x}$$

A large eigenvalue means that the corresponding eigenvector has also large values, which means that this eigenvector encompass most of the variance of the given class. This means we can simply sort the eigenvalues in descending order and pick the largest m out.

Here we want to point out that there exist many different algorithms for computing the eigenvalues, whereas we have focused in this Report in the Power iteration.

3.1 Power Iteration

In mathematics, the power iteration is an eigenvalue algorithm: given a matrix $\mathbf{A}$, the algorithm will produce a number λ (the eigenvalue) and a nonzero vector $\mathbf{v}$ (the eigenvector), such that $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$.

The power iteration algorithm starts with a vector $\mathbf{b}_0$, which may be an approximation to the dominant eigenvector or a random vector. The method is described by the iteration $b_{k+1} = \frac{Ab_k}{\|Ab_k\|}$. So, at every iteration, the vector $\mathbf{b}_k$ is multiplied by the matrix $\mathbf{A}$ and normalized.

Under the assumptions:

- $\mathbf{A}$ has an eigenvalue that is strictly greater in magnitude than its other eigenvalues
- The starting vector $\mathbf{b_0}$ has a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue. then:
- $\mathbf{A}$ subsequence of (b_k) converges to an eigenvector associated with the dominant eigenvalue

Note that the sequence (b_k) does not necessarily converge. It can be shown that: $b_k = e^{i\phi_k} v_1 + r_k$ where: v_1 is an eigenvector associated with the dominant eigenvalue, and $\|r_k\| \to 0$. The presence of the term $e^{i\phi_k}$ implies that (b_k) does not converge unless $e^{i\phi_k} = 1$ Under the two assumptions listed above, the sequence (μ_k) defined by: $\mu_k = \frac{b_k^* A b_k}{b_k^* b_k}$ converges to the dominant eigenvalue.

It needs to said that b_k converges to a multiple of the eigenvector $\mathbf{v}_1$, at rate $\left|\frac{\lambda_2}{\lambda_1}\right|$, where λ_2 is the second dominant eigenvalue. Thus, the method converges slowly if there is an eigenvalue close in magnitude to the dominant eigenvalue.

4 Experiments

Generally speaking, when using real speech data, the dimensionality of the feature vectors it too large to efficiently display in any plot. Therefore the following experiments are artificial and only show how LDA works. We sampled n Gaussians with feature size of 2, which will be reduced to 1 using LDA.

We used Python2.7 with the help of numpy to do the calculations.

In the chosen programming language, the power iteration is implemented as follows:

```
def powerIteration(A):
    b = [random() for i in range(len(A))]
    tmp = [0] * len(A)

    for iteration in range(10000):

        for i in range(0, len(A)):
            tmp[i] = 0
            for j in range(0, len(A)):
                tmp[i] += A[i][j] * b[j]

        normSq = 0
        for k in range(0, len(A)):
            normSq += tmp[k] * tmp[k]
        norm = sqrt(normSq)

        for i in range(len(A)):
            b[i] = tmp[i] / norm

    return b
```

5 Results

References

[1] Tianxing He Lu Chen. Tech-report 2014-4. *Tech-report*, 2014.