SINGLE WORD SPEECH RECOGNITION USING HIDDEN MARKOV MODELS

by

Petr Zadražil

Libor Grafnetr

Course project for Graphical Markov Models
(AE4M33GMM) lectured by

Doc. Dr. Boris Flach,

Ing. Radim Tyleček

Contents

Introduction

Our goal was to perform recognition of single words from a small dataset using Hidden Markov Models (HMM). This task consisted of preprocessing the input data, transforming the data into representation viable for usage in HMM, specifying properties of the HMMs, learning these HMMs and afterwards extracting the learnt knowledge during recognition.

The input data were several series of read numbers from 0 to 30 taken from ShATR Corpus[1]. First, we semi-automatically split the series into single words. The isolated instances were then transformed into sets of Mel Frequency Cepstral Coefficients (MFCC).

From these sets of coefficients we extracted limited number of hidden states using Expectation Maximization of Gaussian Mixture Models (EM GMM). After creating HMM using these hidden states, we performed the training of the probabilities.

Finally, we had trained HMMs that could be used for recognizing unclassified instances. Each of the HMMs is representing one number. The unclassified instance is considered to represent the same number as the model which would generate the instance with the highest probability.

Model description

The approach we chose was to create a separate HMM for each target word class. That is, to recognize numbers from 0 to 30, we created 32 HMMs (0 has two different representations, “zero” and “null”). Observations of the model are cepstrum coefficients representing frames extracted from the input words. Hidden states are generalized representations of the observations ($N_{states} \ll N_{observations}$) . This generalization is realized as Gaussian Mixture Model. This allows for reduction of the number of hidden states to a viable number while minimizing loss of information.

Observations

The input data are samples at a constant sampling rate (in our case 16 kHz). If no transformation would be applied, the model accepting them be would immensely large and sensitive to noise and natural variations, such as voice pitch. Therefore, popular transformation from time domain into frequency domain and consequently into Mel Frequency Cepstrum (MFC) was applied, which facilitates both dimension reduction and partial increase in robustness with regard to noise and natural variations of voice.

Each word was first separated into frames of samples with equal length (25 ms) and ~50% overlap (12 ms) using Hamming windowing function (this gives us ~384 samples per frame). For each frame a MFC was then calculated.

MFCC

The Mel Frequency Cepstrum Coefficients were calculated by using best practices described in European Telecommunications Standards Institute standard[2].

First, frequency power coefficients were calculated by applying Fast Fourier Transform to the samples from the frame. The first half of the power coefficients was then normalized and used later on.

Since human voice mostly occupies only certain part of the frequency spectrum and it desirable to increase resolution in this part of spectrum, transformation from linear frequency scale into Mel Scale is applied. The relation of the scales is given by $m = 2595 \log_{10}\left(\frac{f}{700} + 1\right)$, where f is the hertz frequency

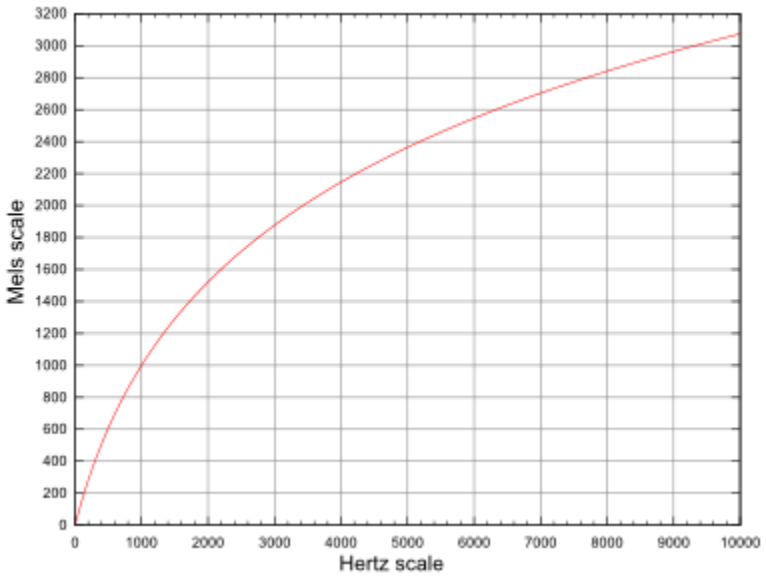


FIGURE 1: RELATION OF HERTZ FREQUENCY SCALE AND MEL SCALE.

Next step is to reduce the dimension of the power coefficients vector. This is done by summing them into much smaller number of bins (we used 24) using triangular window function with 50% overlap.

The two previous steps can in fact be merged into one by calculating transformation matrix that applies both the Mel scale and windowed summation.

At last, the reduced number of binned power coefficients is transformed into log-scale and a Discrete Cosine Transform (DCT) is applied. The properties of DCT allow to further reduce the number of coefficients by half (i.e. to 12 in our experiment).

States

As hidden states for each HMM we used components of a Gaussian Mixture Model trained on cepstrums created from instances of all the dataset. Basically it means that all the models have the same states and only the initial and transition probabilities are different. Number of components of the mixture is subject to parameter optimization described later on. To find locally optimal parameters of the Gaussian Mixture we use our implementation of Expectation Maximization for GMM. Each hidden state is then represented by mean, covariance matrix and weight of a Gaussian distribution in the mixture. The covariance matrix is only diagonal because the number and character of samples does not allow computing meaningful covariances.

The very initial idea was to train Gaussian Mixture Models separately for each number and position. It was supposed to ensure easier transition computation and validation. We were worried about some little noise causing the sample to be matched as a sample of different number. This would lead to zero transition probability in the HMM of the correct number and cause wrong classification. But our dataset was too small to allow this. Furthermore we would get in a serious trouble in case of time-shifted word.

Later we decided to create unique Gaussian Mixture Models for each number. The idea was tested on a simple homogeneous model. Even thought we used whole dataset for both the testing and training the results were below expectation. The best recognition rate achieved was below 0.8 despite the fact we tried as much as 48 states per word. It was mostly caused by the number 1 with very simple model matching some other numbers. Actually the problem was expected and we hope to solve it by some additional multiplication of the result by the accuracy of the Gaussian match. Sadly no good way how to measure accuracy of sample matching Gaussian was discovered.

We also tried to pad the audio signals with zeros to the same length. It would allow us to easily distinguish between short and long numbers but this option was also declined. Our EM algorithm computing Gaussians seems to have trouble with that and most of all it feels like a cowardliness to make the problem easier.

Probability Models

We are using Homogeneous Markov Chain as a model of each number. The model has initial probabilities, transition probabilities and states represented as Gaussians. The model can be one of two variants the “normal” variant and the “strict” one. The difference lays in evaluation. The “normal” version allows one sample to belong to more states concurrently creating distribution between states whereas the “strict” one choose always the state with highest probability for each sample. The second approach is similar to clustering.

We also find a couple of more advanced models widely used in speech recognition as Kalman Filter or the model where each state is represented by a mixture of Gaussians but they seem complicated and hard to implement. And they would also take away the joy of creating custom model which actually does something.

Learning

The learning is separated into two steps. The first step is to create Gaussians. EM algorithm is really a simple one but there are few complications related to implementation.

The biggest issue was numerical stability. While computing next iterations we have to compute with large amount of numbers covering huge range of values. Thus some computations may lead to corrupted covariance matrices. We tried to correct these values modifying Σ matrix (the middle one) from Singular Value Decomposition but the result was often diagonal matrix. Currently we take from the covariance matrix the upper triangular part and the flipped version use as the lower triangular part. After finishing the EM algorithm the covariance parts are stripped out leaving nonzero values only on the main

diagonal. The covariance parts were always nearly zeros, thus not significant, and stripping them makes further computation easier.

Another issue was caused by tiny isolated groups of samples leading into Gaussians with small variances. As our intention was to have Gaussians representing similar amount of samples we modify variances to average values of other Gaussians. For reasonable input data (which our samples are) it solves our problem.

Second part is mostly about computing probabilities. We process all the words of one class together. At first we compute how much each sample matches each state and store it as a matrix. In case of strict model the distribution is updated to contain only zeros and ones.

Then probabilities belonging to initial samples are summed together and normalized to have sum of one. This vector is considered to be our initial probability distribution.

After the transition matrix is computed. It is a square matrix where each value represents transition probability from previous state (row) to the current state (column). The matrix is computed similar way as the initial probabilities. We sum all the transition probabilities between samples and then normalize each row to give the sum one. The only exceptions are “pure ending states” where no transition probabilities can be computed. We leave zeros here letting the recognition algorithm to coop with the situation.

Recognition

As mentioned before we have one model per word class. The model which with highest probability generates such a sequence of samples is considered to classify the given word.

The probabilities for individual models are computed by usual dynamic computing approach. We take initial probabilities and samples as input. In each step we take vector representing probabilities of current state which is element wise multiplied by probability vector obtained from sample. The result represents probability that sequence till current sample was generated by this model. Later the result probability is multiplied by transition matrix and new probabilities of current state are obtained.

We run on to similar issues as with the EM algorithm. The precision problem was solved by normalizing the current state probability distribution to give sum of one. The sum before normalization is remembered for each state. At the end we could simply multiply all the sums and obtain the result. This was not possible in most cases because of computing precision so we were forced to use trick with logarithm. We compute a vector containing logarithm of each value and then compute the mean value. We can use the mean because we have the same number of iterations while computing all the models.

Each sample can have noise which can lead to transition with zero probability. Because of multiplication this one zero transition causes zero matching probability of whole Markov Model. This behavior can be the cause of wrong classification. We introduce parameter “e” which is the minimal probability of each validation algorithm iteration.

Results

We perform series of test in order to optimize parameters. Mostly we tried to find a good pair of the “e” parameter and number of Gaussians. Because the EM part was time consuming we tried every fourth value in interval <4, 52>. In contrary the validation itself was fast so we decided to try 50 different “e” values. We also tried to find out which version performs better, whether the “normal” or the “strict” one. We run 5 fold stratified cross validation on whole dataset.

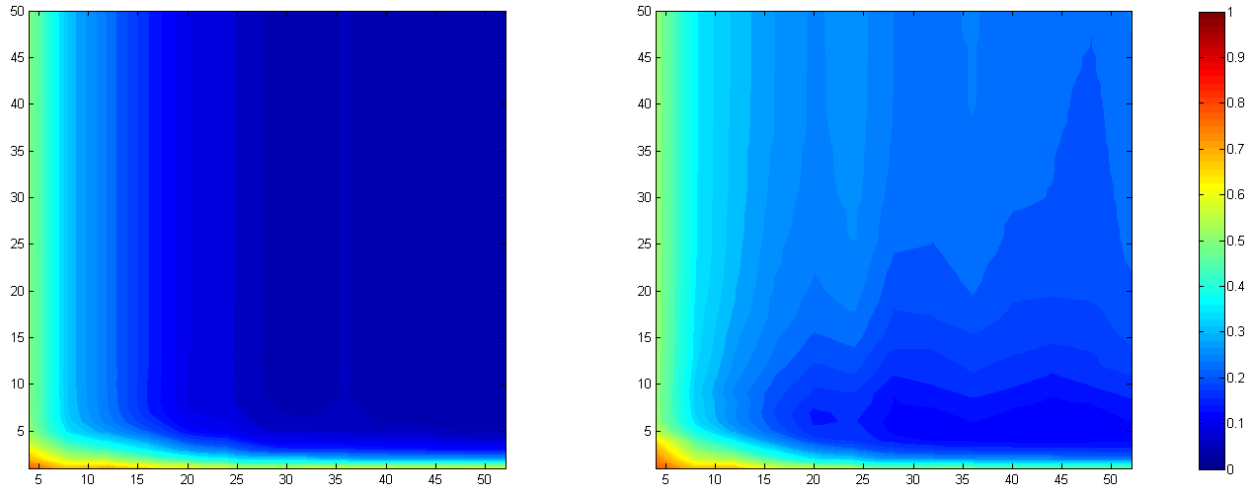
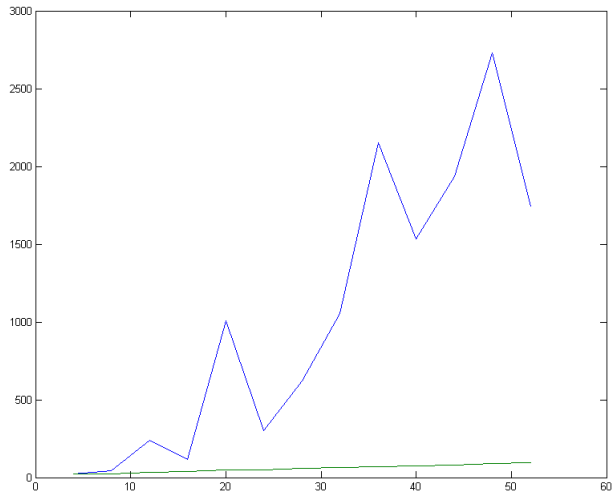


Figure 2: Quality of recognition: Horizontal axis represents the number of Gaussians while vertical axis is representing the “e” value in (2^-Y) scale. The color is rate of wrong classifications. The left image is “normal” mode whereas the right is “strict” mode.



It can be seen, that with increasing number of Gaussians the probability of recognition is rising. Maybe there is a number which makes the models so precise, that recognition of unknown sample would be hard. We didn’t reach such a number because learning these models is time consuming.

The best value was measured for “normal” model with “e” = 2^-13 and 52 Gaussians. The accuracy was 0.958 which was better than expected. “e” smaller than 2^-22 perform slightly worse because there is huge penalization for per wrong transition.

Figure 3: Learning (blue) and validation(red) time in seconds

We also tried our number recognition on sound recorded by us, but we are ashamed of the results. The only number recognized correctly was zero. It was most probably caused by different speaker, bad recording device and conditions. Our algorithm considers most of number to be 23, 11 or 0.

Conclusion

We have successfully implemented, trained and tested single word speech recognition in Matlab. We also sorted out all the troubles with numerical stability and once again experienced the difference between theory and implementation.

Our results were better than we expected, however we believe there is still a lot to improve. It is clear that further enhancement requires more samples from different speakers and/or different kind of preprocessing.

---

[1] The ShATR Multiple Simultaneous Speaker Corpus - http://www.dcs.shef.ac.uk/spandh/projects/shatrweb/index.html

[2] European Telecommunications Standards Institute, ETSI ES 201 108 standard, Front-end feature extraction algorithm

[3] Rabiner, L., R. A Tutorial on Hidden Markov Models and Selected Application in Speech Recognition