ALFANUM SYSTEM FOR CONTINUOUS SPEECH RECOGNITION

Darko Pekar, Radovan Obradović, Vlado Delić
School of Engineering, University of Novi Sad, Yugoslavia
e-mail: pekard@eunet.yu

Abstract

This paper shortly presents program package for continuous speech recognition which is, so far, successful with small and medium dictionaries. Package is very large because it contains both modules for training and recognition. Each of these modules consists of several submodules and variety of classes and functions. It includes two libraries developed in last two years by the same authors. Those are **slib** library for digital signal processing [1] and general purpose **an_misc** library. Program package is product of several years of work on the automatic speech recognition topic, starting from isolated words recognition, over connected words recognition, to continuous speech recognition using phoneme in context, which this system is based on. Since the system is based on phoneme in context recognition, it supports recognition of any set of words (grammar) which needs to be recognized. Changing grammar requires no additional training or speech database recording, but only building of a new trellis, which takes few seconds. The whole program is written in C++ programming language, it is fully developed by the authors, which means that it does not rely on any third party specialized library. Software is in its largest part independent of the platform or the operating system (except for the part which requires communication with hardware).

INTRODUCTION

Because of the size of the package, this paper will be divided into several parts, although even then there won't be enough space to fully describe all aspects of the system functioning. Tutorial will be given through the phases of training and testing which should be followed in the process of creating one ASR system. These phases comprise:

- creation and/or adjustment of the global **ini** file,
- feature extraction,
- label file creation,
- initial training,
- ML training,
- grammar setting,
- recognition.

1. GENERAL INFORMATION

1.1 Configuration (ini) file

In this text file user can define all the parameters which are important both for training and recognition. It is not classical **ini** file, it has somewhat different structure. List and vector structures are supported, so we can define data structure of arbitrary complexity. This is enabled by class *sobj* as well as appropriate parser. Here is a short example of how can data look like in **ini** file:

```
{
// features data
   samples_per_second = 16000.0,
   window_duration_ms = 30.0,
   frame_duration_ms = 10.0,

   derivatives_data =
   [
      { window_ms = 30.0, weight = 2.0 }
      { window_ms = 50.0, weight = 2.0 }
   ]
   cache_dynamic_features = 0 // false
}
```

There are three levels of parameters configuring. The first is the *default* level. These values are defined in the source code and are diplayed every time the program is started. The second level is **ini** file in which we can override any *default* value. The third level is command line in which we can define new values, which will override those from the previous levels. For example, if command line looks like this:

```
csr test -cdf 1
```

switch **-cdf** enables setting variable **cache_dynamic_features** to 1, even it has been set to 0 in config file. Not all of the variables have the ability to be altered from the command line (switches aren't provided), only those for which it makes sense.

1.2 Feature extraction

Feature extraction is based on previously mentioned library **slib** and its script version **slib_scipt**. Feature extractor is given in textual form, where are described all the blocks used for feature calculation. User can choose among wide range of usually used features such as: cepstral coefficients, (several kinds), energy and log energy, zero crossing rate, pitch, degre of voicing, as well as derivatives of these static features. So, features are not fixed in source code, and user has oportunity to choose desired features. Since it is rather complex to create whole script file for feature extraction, several most used combinations are offered. User can use those files or modify them to match his need.

Program supports so-called feature caching on hard disk. It is well known that the process of feature extraction is rather time consuming task, but when these features are once calculated it is usually unnecessary to alter them during

training and testing. Therefore it is convenient to store extracted features to disk and simply to load them in the next iteration of training. This task is performed automaticaly, but the user can turn it off. Features are cached in directory which name is the function of extractor name and its content. Every "wav" file has corresponding cached ("mfc") file. Directory structure is the same as the one of the speech database. We emphasise that the name of this directory is function not only of the name of the used extractor but also of its content (*hash* value is calculated), as well as several other parameters which can influence features shape, so it is impossible for program to load cached features if user changed something in the way they are calculated. All these files and directories are stored in directory "cache", so that unnecessary files can be manualy deleted.

It is interesting that the features are being saved on disk in format which is compatible with HTK [4] parameter files. This is done delibarately because of the two things. Firstly we wanted to check how do our features function on the system trained with HTK. We also wanted to see eventual advantages or drawbacks of our algorithm for phoneme in context training compared to HTK.

Delta features can be calculated in two ways. They can be calculated directly, in extractor, using **slib** blocks, which is prefered approach if we want to use our system in some *on-line* recognition application (meaning that there isn't whole file at disposal, but the samples gradually arrive). Second approach is to calculate them afterwards in main program. If the user wants this, he should define the number of derivatives and each derivative window size in **ini** file. This approach can save disk space if we are using caching, because the files will be three times smaller (if two derivatives are used). In the second case derivatives are being calculated "on the fly", which is not time consuming.

Energy normalization can also be performed in two ways, similar to derivatives calculation. If we need *on-line* recognizer, **slib** blocks should be used, when normalization is performed in so-called moving window (**SMovingMax** and **SMovingMin** blocks) with some, configurable delay. If we process signal in file, much better approach if file level normalization. This is also specified in **ini** file.

Well known method for robust, channel independent, cepstral coefficients is supported, so-called *cepstral mean subtraction* [6]. It can also be performed "on the fly" or on the file level.

After extraction features can be dividen into several data streams. Which features would belong to which stream and how many of them is going to be is totaly arbitrary, and is given in **ini** file. For building continuous density HMM systems, this facility is of limited use and by far the most common case is that of a single data stream. However, when building tied-mixture systems or when using vector quantisation, a more uniform coverage of the acoustic space is obtained by separating energy, deltas, etc., into separate streams.

In the same structure as the data streams, feature weights are given. This information is very important in initial training and vector quantisation, but once the variances are calculated it becomes irrelevant.

1.3 Model structure

Our system uses so-called *semi continuous HMM* [6] approach, which means the following: certain phoneme in context models don't have their, and only their, mixtures, but there is mixture repository which contains all the mixtures obtained in training, and every model has a list of refereces to the belonging mixtures. This approach offers great time savings during recognition, because the number of mixtures to be calculated is greatly reduced (depending on the grammar).

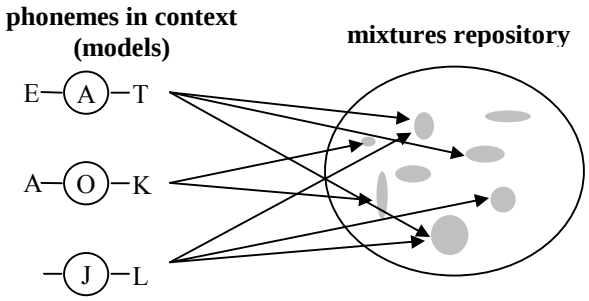

Figure 1.1: Model structure

Although the mixtures are shared among models, each model has its own mixture weights for all mixtures it uses. Each model's duration distribution which was present in the speech database is also calculated. Duration modeling during recognition is based on this information. Furthermore, models contain mark of the phoneme they represent, as well as optional left and right contexts. If the model has left context, then it can be used only in that context, in other words only after that phoneme. The same goes for right context. Therefore if the model doesn't have neither left nor right context then it is so-called monophone which can be used in any context. Of course, during trellis building, always is taken the best model that exists, the one with the widest context fitting the required.

Due to perspicuity, we talked about phonemes so far, although those are actually subphonemes. Other than using term "subphoneme" we can use "state", since they are synonyms. For example, phoneme "A" can consist of three subphonemes "A0", "A1" and "A2". Each subphoneme has its own model, with corresponding contexts. This enables that, in the lack of sufficient number of observations in training base, for example for left subphonemes ("A0") only left contexts are being made, and for the right subphonemes only the right ones. It is usual to make right contexts for the middle subphonemes, since the effects of nasalization and voicing equalization usually go backward, and not forward. If there are enough observations, triphones will be made. Number of subphonemes for each phoneme is defined in configuration file.

1.4 Label files

Training procedure requires, besides audio files with recorded sequences, file containing information about what was said in that audio file and where the boundaries between the speech units are. Therefore, since this file contains information about labels, the usual name is label file.

One line in our label file should look like:

```
"pera.wav" "gender=male;" "_ Po Pe E R A spk _ #" 0.0 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Three wholes can be seen:

1. The name of the audio file is given under quotes.
2. **Class** field. Can contain arbitrary number of data given in *key=value* form. It can be used for speaker data, information about file itself (quality etc.), about whether the file was inspected etc.
3. Labels, i.e. pronunced phonemes. The programme does not restrict this to be strictly phonemes. As can be seen stop "P" is divided into closure and explosion. In the same way, the whole word could be put under one label. The only reserved marks are "_", which is sign for silence and "#" which is sign for final phoneme ending. We can also see "spk" mark which can mean some noise originating from the speaker, but it is all up to program user to define and use such marks. Another restriction is that label shouldn't end with a digit (e.g. A1), because these marks are used internally for phoneme division into subphonemes.
4. Borders, beginings to be exact, of corresponding phonemes, given in seconds.

Every line in label file gives information about one audio file, so the number of lines and number of audio files processed is the same.

2. MODEL TRAINING

2.1 Initial training

As mentioned before, *semi continuous HMM* model is used, i.e. tied mixtures approach. The same mixture can be used by any model, and which models will actually share the mixtures depends on initial training. There is not much sense to share some mixture between, for example, some fricative and a vowel. Therefore so-called phoneme groups are defined within which phonemes can share mixtures. Each group is disjoint with other groups. These groups can be defined in **ini** file. There is another approach in which each subphoneme represents separate group. For example, all sequences representing "A0" will be stored in one group, and certain models (i.e. "A0" in different contexts) will use appropriate mixtures from that repository. Although this second approach makes more mixtures, it much more successful in recognition accuracy.

Descriptive version of the initial training algorithm is given in the following.

```
main
{
    Extract features and all contexts available in the base
    For each phoneme group:
    {
        Collect all vectors describing any of the phonemes in group
        Vector quantisation (making mixture repository)
    For each subphoneme in group:
    {
        Collect all vectors describing that subphoneme
        MakeMoniphone(vectors, subphoneme)
    }
    }
}
MakeMonophone(vectors, subphoneme)
{
    If it is left subphoneme (e.g. A0 from A0 A1 A2)
    {
        For all left contexts available in the base:
        {
            Move all vectors concerning that context
            If number of moved vectors is bigger than defined for biphonemes and number of remaining is bigger than defined for monophones
            {
                MakeLeftBiphoneme(moved vectors, left context)
            }
            else
            {
                Move vectors back
            }
        }
    }
    else (middle or right subphoneme)
    {
        For all right contexts available in the base:
        {
            Move all vectors concerning that context
            If number of moved vectors is bigger than defined for biphonemes and number of remaining is bigger than defined for monophones
            {
                MakeRightBiphoneme(moved vectors, right contexts)
            }
            else
            {
                Move vectors back
            }
        }
    }
    MakeInitialModel(remaining vectors) - monophone
}
MakeLeftBiphoneme(vectors, left context)
{
    For all right contexts available in the base:
    {
        Move all vectors concerning that context
        If number of moved vectors is bigger than defined for triphonemes and number of remaining is bigger than defined for biphonemes
        {
        MakeInitialModel(moved  vectors) - triphoneme
        }
        else
        {
            Move vectors back
        }
    }
```

```
    MakeInitialModel(remaining vectors) - biphoneme
}
MakeRightBiphoneme(vectors, right context)
{
    ...
}
MakeInitialModel(vectors)
{
    Collect all the mixtures containing at least one vector
    Calculate average probability density function value
    (pdfv) of vectors for all mixtures
    while (pdfv > defined value)
    {
        Expel mixture with smallest number of vectors, and
        those vectors join other, the closest, mixtures
        Calculate average probability density function value
        (pdfv) of vectors for all mixtures
    }
    Make model from the remaining mixtures
}
```

In the last step when the remaining mixtures are used to make a model, initial mixtures weights are set proportional to number of vectors left in that mixture. It is not written in the algorithm, but in the same procedure duration distributions for each model are being calculated. Actual distribution of its duration in the training database is memorized, which will be used during recognition.

Even when we restrict phoneme groups to only one subphoneme, still relatively unlogical mixture sharing can arise. For example, it doesn't make much sense to share mixtures between model S-A0 ("A0" with left context "S") and model E-A0. It makes sense to share mixtures between those models which have similar context. Meaning that in one group should go contexts made of vowels, in other fricatives etc. In HTK this problem is solved with so-called *tree-based-clustering* algorithm, but our system doesn't treat this problem explicitly. However if we adjust feature weights adequatly, very cosistent mixture sharing among models can be obtained. Therefore great weights should be given to following features: energy, two first cepstrums (rough spectrum description which can efficiently distinguish fricatives, vowels and nasals) and their derivatives.

Procedure of initial training is very simple from the aspect of programme user. User should provide the following:

1. Speech database given as audio files (besides *wav* format, *adc* and *raw* are supported).
2. Label file with phoneme boundaries. We shall see later that the procedure of bound setting can be greatly automated.
3. Configuration file (actually, only existing should be modified).
4. Feature extractor (or use some offered).

Initial training will make so-called *dictionary* file whose *default* name is "schmm.dict". It contains all data about built models, but also the contents of used extractor script and configuration file, so latter, during recognition, they aren't needed. As it could be concluded from the algorithm, the program will build all the triphonemes with sufficient number of occurences in base, all biphonemes, and from the remaining vectors all monophones.

In this way set of models is obtained which can be used for building arbitrary trellis, regardless of which phoneme sequence is required.

2.2 ML training

In all ASR systems it is common to have some form of *maximum likelihood* training after initial. In this system *expectation maximization* algorithm is used [8]. The algorithm is described in detail in mentioned literature, but it is algorithm for CDHMM model, i.e. the case without mixture sharing. Our algorithm is, of course, adapted to applied SCDHMM model.

3. RECOGNITION

For the purpose of recognition the following should be provided:

1. *dictionary* file which contains HMM models, obtained by initial and ML training.
2. Recognition grammar, i.e. graph of possible words and allowed transitions.
3. Phonetic transcriptor
4. Postprocessor

3.1 Acoustical level

As mentioned, tied mixtures model is used. As far as state transitions, only sequential model is supported so far, so it is possible to transfer only to the next state. This is the case when speaking about states within one phoneme. Transfers between different phonemes and words can be arbitrarily complex, which will be seen later.

Standard Gaussian mixtures are used with diagonal covarianse matrix. For state probabilities calculation two approaches are used [2]: calculation over weighted sum of Gaussian mixtures, and *winner take all* method. Second approach has, once again, proved equally good compared to the first one, and provides significant time savings.

3.2 State duration modeling

After state probability calculation in certain time instants, optimal sequence is found by Viterbi algorithm. One, well known, method for modeling state durations is over probability for staying in current state and probability for transition to the next state:

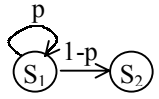

Figure 3.1: Classical state duration modeling

It is also known that this model gives exponential distribution of state durations that doesn't match real, natural distribution which would optimally model phoneme durations.

Therefore we supported, besides this classical approach, model which optimally models state durations. Suppose state S_1 had following duration distribution in training database:

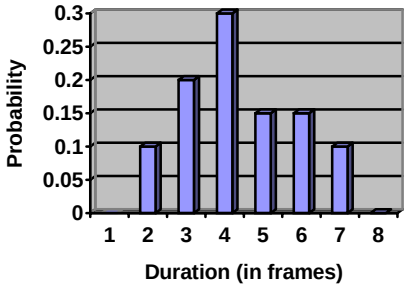

Figure 3.2: Duration distribution of state S_1

Then it could be divided into several states:

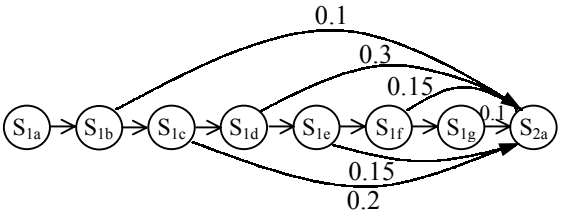

Figure 3.3: Expanded model for duration modeling

Unmarked transition probabilities (those within same macro state) are 1. It is obvious that this model acquires the very distribution we had in training database. However it is also obvious that number of states is drastically increased (for some states up to several dozen times). These are the same states, from acoustic point of view, but nevertheless significantly burden Viterbi algorithm and lag recognition. By using this model certain accuracy improvement is gained, but accuracy vs. CPU time ratio is not very well. Hence it should be used when CPU power is not a problem.
Transition graph shown in figure 4 strictly restricts state S_1 duration from 2-7 frames. In case of relatively small training database these strict constraints can lead to poor recognition results, if some duration wasn't present in the base. Therefore we left possibility for the state to last less than minimal and more than maximal number of frames, but probabilities for such events are set to very small values.
Although better than standard, this method still doesn't treat another aspect of state duration modeling. Here is not modeled joint state probability distribution, and they are treated as statistically independent. It known that this is wrong.
In the end the overall probability of certain state is calculated as:

$$P = P_t\left(T = N \mid S_1\right)^K \prod_{n=1}^{N} P_a\left(S_1 \mid t = n\right) \qquad (3.1)$$

Where P_t is probability that the state S_1 lasted N frames, and P_a is acoustical probability of state S_1 in instant n. If K would equal 1, this would be classical case, in accordance with probability theory. However, if this coefficient would be given some values bigger than 1, it would lift weight of duration probability compared to acoustical, and the experiments showed that this can lead to recognition accuracy improvement. This coefficient can be set in configuration file.

3.3 Grammar

Our system can be used as small and medium vocabulary speech recognizer. In these cases it is usual to use regular grammars. Grammar can be given in two ways:

1. By using transition diagram
2. Through EBNF form (*Extended Backus Naur Form*) [4].

3.3.1 Setting grammar by using transition diagram

Suppose we want to define simple grammar which will recognize arbitrary number of digits. Graphical representation of appropriate transition diagram would look like:

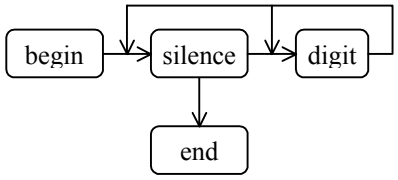

Figure 3.4: Transition diagram for digits grammar

So both direct transitions from digit to digit, and transitions over silence state are allowed. It is very important to define this in order to inform program to make optimal models. If only transitions over silence would be enabled, it would result in shorter and faster trellis, but the system accuracy would be considerably degraded if the speaker wouldn't make explicit pauses between words.
This diagram is given in textual file and could look like this:

```
main
{
  begin -> silence;
  tisina -> digit;
  digit -> digit;
  silence -> end;
}
cifra
{
  begin -> nula;
  begin -> jedan;
  nula -> end;
  jedan -> end;
}
tisina { "_" }
nula { "NULA" }
jedan { "JEDAN" }
```

Identifiers "digit", "silence", "nula" and "jedan" are so-called rules and are represented with either new transition diagram (as in "digit") or text sequence they represent. Reserved identifier "main" describes the main rule in grammar. Reserved identifiers "begin" and "end" define begining and end of each transition diagram.

3.3.2 Setting grammar by using EBNF

This second method is by far more used and a file which describes it is much smaller. Let's say that we want to define grammar for room choice by number. Suppose we want to allow some introduction sequences such as: "Molim Vas", "Molio bih" and "Treba mi". This is done in following way:

```
intro = (MOLIM VAS)|(MOLIO BIH)|(TREBA MI);
```

After that user can say the word "Sobu" (room) followed by array of digits. Digit definition (0-3):

```
digit = NULA | JEDAN | DVA | TRI;
```

Final grammar would look like:

```
main = [gar][$intro][SOBU]<$digit>[gar];
```

Sqare brackets mean that it is optional sequence, and "<>" brackets mean that the sequence is repeated one or more times. "gar" is short of *garbage*, but it is not reserved word. This should be model which is trained on noises which were in training base and serves to absorb such occurences in recognized speech.

Among already seen marks: "|", "[]" and "<>", we can use "{}" brackets which mean that the sequence is repeated zero or more times. Once again it is obvious that the convention is compatible with HTK, only "<<" and ">>" symbols are not supported.

EBNF parser internaly translates this compact record into transition diagram, and further into complete trellis.

It should be observed that in grammar we didn't explicitly define silence existance, neither should the transitions go directly or over silence states (i.e. is it requested that speaker makes explicit pauses). By setting the appropriate parameter in **ini** file, recognizer can be told to do this automaticaly. So transition between any two words can be direct or over silence. This option gratly relieves user.

3.4 Phonetic transcriptor

It could be noted that in grammar creating we use ortographic transcription of the words to be recognized. In Serbian language ortographic and phonetic transcriptions are almost the same (although even there are differences), but for the majority of languages this is necessary step. Phonetic trascription is conducted in two steps:

1. Lookup for the word in pronunciation dictionary.
2. Application of set of rules on ortographic transcription.

3.4.1 Word lookup

If the user wants to use this option he should provide file with pairs of ortographic and phonetic transcriptions. For example:

```
NULA    N U L A
JEDAN   J E Do De A N
JEDAN   J E A N
```

Each line describes new word. In phonetic trascription phonemes are separated with space. It can be seen that multiple pronunciation of the same ortographic word is supported. These cases are, later, on the grammar level separated into independent branches for recognition. During detranscription they are returned to the same ortographic transcription, regardless of which version of pronunciation algorithm recognized.

From the given example of the pronunciation dictionary we can see that we separated stops (and africates) into closure and explosion, which is possible but not mandatory approach.

3.4.2 Set of rules application

If we are talking about language with very strict rules of ortographic to phonetic conversion, foregoing approach is too spacious and demands unnecessary making of large pronunciation file. In these cases it is more convenient to apply certain set of rules to ortographic text and obtain phonetic transcription. Here is how it could look like for Serbian language (short version):

```
vowels { A E I O U }
rules
{
  // majica -> maica
  vowel J I -> vowel I;
  // petnaest -> petnajst
  N A E S T -> N A J S T;
  // separating stops and affricates
  // into closure and explosion
  P -> Po Pe;
  C -> Co Ce;
  ...
}
```

All geven rules are applied to word in ortographic transcription in a given order (important in some cases). With such set of rules even some more complicated languages could be described, and exceptions could be put into pronunciation dictionary. And that's how it works: if the word wasn't in the pronunciation dictionary, rules are applied to it.

One drawbac of this approach is that it doesn't support multiple pronunciations of the same word. If we want that dictionary must be used.

3.5 Detranscription

Detranscription is also an interesting process following the recognition. Suppose recognizer gives following sequence:

```
N U L A J E A N Do De V A
```

Of course, user will be interested in words in the original, ortographic transcription, conveniently separated, i.e.:

```
NULA JEDAN DVA
```

In order to obtain this sequence it is necessary to apply set of rules to phonetic trascription, which could, in this case, look like:

```
rules
{
   J E Do De A N -> "JEDAN ";
   Do De V A -> "DVA ";
   J E A N -> "JEDAN ";
   N U L A -> "NULA ";
}
```

There is a little problem here if we have word which is subset of the bigger one (e.g. if we had word "JE" in this case). If that "smaller" rule is applied first to word, it could leave part of word in phonetic state which will lead to error. That is why the preceding rules are sorted by the size of the phonetic part.
This set of rules is generated automaticaly, during grammar transcription, so user need not to worry about that.

3.6 Postprocesor

After detranscription one possible result would be:

```
_ NULA _ gar _ JEDAN _ gar _
```

User usually isn't interested in information about where the silences were and did the noises occur, so he would have to clean received sequence on his own. Therefore we provided option which does this before returning result to the user application. User should only define few simple rules which would do that, and in this case the could look like:

```
rules
{
   _ ->;
   gar ->;
}
```

4. AUTOMATIC LABELING

As once mentionet, part of the programme for training requires presence of labeled audio sequences, i.e. label file with correctly set boundarioes between speech units. It is obvious that this would require huge amount of work if all done by hand. Therefor we provided option for automatic labeling, if some initial acoustic models are provided (*dictionary* file) and a label file with correct phonetic transcription. This is done by building trellis which would force recognizer to run through all phonemes in label, and then let Viterbi algorithm to find the optimal path. Obtained segmentation is written into the file.
Of course, it isn't hard to notice "small" paradox: where did we get *dictionary* file? Well, some initial labeling would have to be done by hand. By using relatively small number of labels done by hand some initial model can be built. It will be relatively poor, but can serve to automate process of labeling in the next iteration. Then we can proceed in several ways.
One way is to perform automatic labeling on the whole base, and then to train the system with those labels, discarding the outliers during the training. In the next iteration we do the same, but with the new model.
Second approach is to notify the usual problems after first iteration of automatic labelation, i.e. spots where programme makes mistakes. Then label by hand another part of base with sequences containing problematic phoneme combinations. Then train the model on the extended subset etc. Some combination of these two approaches is possible.
During automatic labeling several details should be observed:

- Models should have small number of mixtures (not more than two).
- It is wise to decrease *frame_duration* parameter in order to obtain preciser segmentation.
- Speakers should be divided into several groups, at least by gender.
- For duration modeling it is better to use second approach, using real duration probability.

5. IN THE RESEARCH PHASE

All mentiond methods are completely debuged, tested and operational. Besides those there are several more ways of training and recognition which hadn't been fully implemented or sufficiently tested until now. Nevertheless we will name the few.

5.1 Corrective training

Corrective or discriminative training should additionaly improve modles obtained by ML training, in sense that they are more pulled apart, and to make better distiction between different models. Used algorithm is very similar to that applied in [2], with the difference that we compare phonemes in context, and not complete words. This leads to number of new problems:

1. We can't make distinction between two models, if they have subphonemes with the sam origin as their basis (e.g. A0 and A1).
2. Even if basis phonemes don't have the same origin some phoneme can be very similar to another. For example model **A2**+M (right subphoneme from **A** whose right context is **M**) and model A-**M0** (left subphoneme from **M** whose left context is **A**) look alike very much and describe parts of signal which often overlap.
3. Closures of all unvoiced stops and africates have practicaly the same features. The same goes for closures of voiced consonants.
4. Depending on channel and a speaker many other phonemes can look alike. For example in telephone channel, voiced - unvoiced consonant pairs are very similar due to the lack of lower 300Hz. If the speaker pronunced in some atipical and/or mellow way, some phonemes could look alike for which we would never say that they are similar. For example, "D" in mellowly pronunced word "JEDAN", can look like "R".
5. When all mentioned considered we come to a conclusion that it is sensible to make additional distincion only between models which already differ significantly, which puts question over the whole idea.

Due to given reasons it will be necessary to reinvestigate phoneme level distinction approach. Maybe it will be implemented for certain grammars, i.e. make distinction between words, which makes more sense.

5.2 Word spotting or wild-cards usage

In this terminology wild-card would mean model which could describe any phoneme, sufficiently well, but its acoustic probability should be less than that of the correct model. If we would have such moedl at a disposal we could make the following grammar:

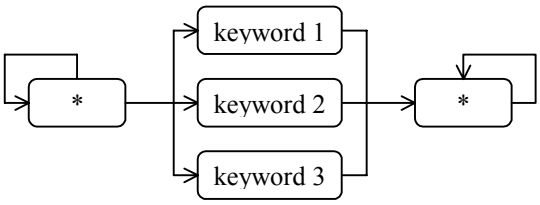

Figure 5.1: Transition diagram of the *word-spotter*

If any of the keywords is pronunced in test sequence, optimal path through such grammar would be over that keyword and it would be recognized ("spotted"). If no keyword is spoken, the best probability would have staying in wild-card (*) state.
Option which makes such models is implemented, but that part is still in testing phase.

CONCLUSION

In this paper we described one complete programme package for continuous speech recognition, based on phoneme in context recognition. Many standard methods for training and recognition are applied, and some new are tested, which represent author's scientific contribution to this field. In many ways this package is compatible or even similar to well known HTK. Results obtained by using this package are not only comparable to those obtained by HTK, but surpass them. The whole package is written in C++ programming language, in comprehensible and elegant form, but also highly optimized, where necessary, so it is ready for commercial use.

LITERATURE

[1] D. Pekar, R. Obradović, *C++ Library for Digital Signal Processing - slib*, Telfor, Belgrade, November 2001.

[2] D. Pekar, R. Obradović, V. Delić, *Connected Words Recognition*, Telfor, Belgrade, November 2001.

[3] R. Obradović, D. Pekar, S. Krčo, V. Delić, V. Šenk, *A Robust Speaker Independent CPU Based Speech Recognition System*, *Eurospeech*, Vol.6. pp. 2881-2884, Budapest, September 1999.

[4] S. Young, D. Kershaw, J. Odell, D. Ollason, V. Valtchev, P. Woodland, *The HTK Book*, 1995-1999, Microsoft Corporation

[5] Rabiner, B.H. Juang, *Fundamentals of Speech Recognition,* Prentice-Hall, New Jersey, 1993.

[6] Mikko Kurimo, *Application of Learning Vector Quantization and Self-Organizing Maps for training continuous density and semi-continuous Markov models*, Licentiate's Thesis, Helsinki University of Technology, 1994.

[7] J.C. Junqua, J.P. Haton, *Robustness In Automatic Speech Recognition,* Kluwer, 1993.

[8] Perry Moerland, *Mixture Models for Unsupervised and Supervised Learning*, Ph.D. thesis, Swiss Federal Institute of Technology, 2000.

[9] S. Krčo *Apendage to the Methods of Isolated Words Recognition,* ETF, University of Belgrade, 1997.

[10] N. Draper, H. Smith, *Applied Regression Analysis,* Wiley, 1981.

[11] T. Back, H. P. Schwefel, *An Overview of Evolutionary Algorithms for Parameter Optimization,* University of Dortmund, 1993.

[12] P. Yao, R. Leinecker, *Visual C++ 6 Bible,* IDG Books Worldwide, Foster City, USA, 1999.

[13] A. Stepanov, *Standard Template Library,* Hewlet Packard, 1995.