EVA[☆] - A toy speech recognition system

Eduart Uzeir

Università degli studi di Bologna

Abstract

In this paper we present EVA, a toy speech recognition system developed over the well-known open-source Kaldi ASR[1] toolkit. The idea behind EVA is to study and develop a very small virtual assistant system based on natural speech recognition. In the present days this kind of systems and the technologies empowering them are vastly available on the market under different names and capabilities. Honorable mentions are the Amazon's Alexa[2] and Google Home[3]. The development process is described in an exhaustive step by step manner that starts with the data acquisition and ends with the performance evaluation of the system.

Keywords: Natural Language Processing, Speech Recognition, Kaldi[1]

1. Introduction

Nowadays we are assisting an ongoing revolution on the field of Artificial Intelligence triggered by the vast amount of available data and the enormous computational power of today's computers. Natural Language Processing as a very promising and fertile sub-field of AI is experiencing a renewed interest by the researchers and companies allover the World. In Dan Jurafsky's[4] words: "This is a particularly exciting time to be working in Natural Language Processing. The vast amount of data on the web and social media have made it possible to build fantastic new applications".

[☆]Eduart's Virtual Assistant
[1]`kaldi-asr.org`
[2]`https://developer.amazon.com/alexa`
[3]`https://store.google.com/it/product/google_home`
[4]`https://web.stanford.edu/~jurafsky/`

This new applications extends form the Q&A systems like IBM's Watson[5] to the sentiment analysis and live session machine translation systems. As mentioned previously, the very large amount of data available on the Internet has made possible the construction of enormous datasets of written and spoken language called Corpora in the natural language processing terminology. The use of this resources in combination with the new technologies in the field of Machine Learning has produce very efficient and reliable systems concerning with language processing. Neural Networks in various shapes and flavours are currently the best choice on the task of training and testing language models. Chomsky's criticism toward the early empiricist approach on the linguistics leads to new ideas and new responses that tried to level down the contradictions. As we know, one of the most insistent reasons why Chomsky disapproved the empiricists approach was the impossibility of artificial Corpora to capture the real extent of a natural language. The general accepted idea is that a natural language is a living thing that evolves over time, and this is the reason why in a finite size Corpora the data will be always skewed. The limitations of the Corpora are theoretically expressed by the Zipf's[6] law. Zipf's[2] law is an empirical observation that relates the size of an organization with it's rank. In the case of natural languages Zipf's law state that the frequency of a word in a Corpus is inversely proportional with the rank of that word in the frequency table. This observation has as an immediate consequence the "data sparsity" that we see in our Corpora. Zipf's law that characterized the English language can be expressed with the following mathematical formulation.

$$f(k; s, N) = \frac{1}{k^s H_{N,s}} \tag{1}$$

The variable "k" represent the rank, "s" is a constant that characterize the distribution (s=1 in our case) and "H" is the Nth harmonic number.

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{N} = \sum_{i=1}^{N} \frac{1}{i} \tag{2}$$

. Here "N" represent the number of words in the English language.

[5]`https://www.ibm.com/watson/`

[6]`https://en.wikipedia.org/wiki/Zipf's_law`

Figure 1 shows the result of the application of the Zipf's law on a set of the most frequently used words in the English language. Just by looking on the chart one can see the big distance between adjacent words.

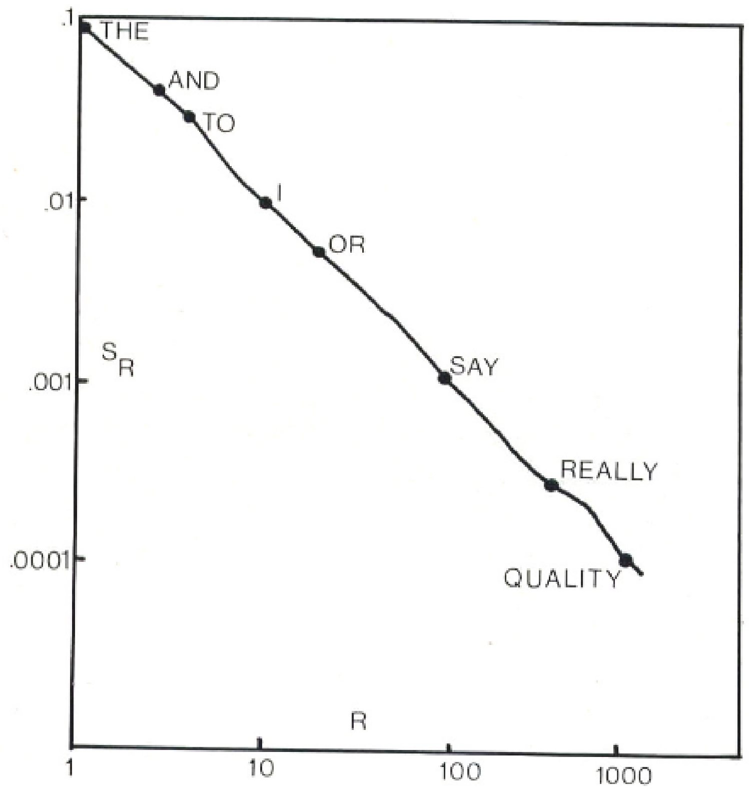

Figure 1: Frequency of word usage in English

Even though the contemporaneous Corpora contain billions of words and sentences the Zipf's law is still valid and obligate us to enrich further more our Corpora in order to obtain better approximations of a given language. In this project we are going to use the Kaldi, an open source speech recognition tool that is very handy in the process of creating a speech recognition system and is utilized mainly by the research community. The first step is to install Kaldi and the other software needed and then prepare the dataset, that as we know is the most important part in process of creating a model. Once we have prepare the dataset, Kaldi will deal with the process of training and testing our data. We will create different models on different datasets in order to be able to experiment the potential of the system. At the end we will present the performance achieved by the system.

2. Dataset

Like all the other systems based on machine learning technology Kaldi use a dataset of audio samples in order to extract and learn input's features. In this sense the size and the quality of the input data is of extreme importance. In this section we are going the describe the process of data acquisition and preprocessing.

2.1. Data Acquisition

EVA speech recognition system contains two datasets.

- General sentences (200 utterances)
- Arithmetic sentences (800 utterances)

The first dataset contain 200 utterances spoken by 10 different participants, 5 males and 5 females. Each of the participant had to perform 20 sentences in natural speaking voice. Table 1 show the list of utterances.

List of general utterances	
01. ***Hello Edward***	11. ***When is my birthday***
02. ***What is your name***	12. ***Where i live***
03. ***What time is now***	13. ***Say my name***
04. ***Call my wife***	14. ***Turn off the light***
05. ***Switch on the light***	15. ***Check my agenda***
06. ***Read my book***	16. ***Read my email***
07. ***Play some music***	17. ***Send an email to my professor***
08. ***How is the weather today***	18. ***Goodbye Edward***
09. ***Order a pizza for me***	19. ***Dim the light 20 percent***
10. ***What is my name***	20. ***Goodbye***

Table 1: List of general utterances

We asked 10 fluent English speaking participants to use Telegram or WhatsApp and send us the 20 utterances as 20 separate audio files in one burst. Is important to mention that the audio files were not recorded previously but on the spot. The received files were in Opus (.ogg) format and had to be further elaborate, a process that we are going to clarify later on. The second dataset contains 800 spoken sentences over the basic arithmetical operations (PLUS, MINUS, TIMES, OVER). In this case the participants were 2, one for each gender. Each participant has been asked to perform 100 utterances for each operation, 400 in total. Here is a short list of the utterances in a synthesized tabular form for the PLUS and MINUS operations.

List of arithmetic utterances for PLUS and MINUS	
01. ***Zero PLUS Zero***	101. ***Zero MINUS Zero***
02. ***Zero PLUS One***	102. ***Zero MINUS One***
... ***. . .***	... ***. . .***
10. ***Zero PLUS Nine***	110. ***Zero MINUS Nine***
11. ***One PLUS Zero***	111. ***One MINUS Zero***
... ***. . .***	... ***. . .***
20. ***One PLUS Nine***	120. ***One MINUS Nine***
... ***. . .***	... ***. . .***
... ***. . .***	... ***. . .***
100. ***Nine PLUS Nine***	200. ***Nine MINUS Nine***

Table 2: List of arithmetic utterances (+, -)

Table 3 show the utterances for the other 2 operations TIMES and OVER.

All the audio files are organized in separate folders named after the participant name (only the initials of the first and last name have been taken in consideration). We used the following naming convention for all the audio file.

UserInitYear_RecordingDate_Gender_FileNumber.Codec

UserInitYear is the name of the participant and the year is in all the files 2018, *RecordingDate* is the exact date when we received the audio files, *Gender* as implied by the name represent the gender of the participant and *FileNumber* is a progressive number in the interval [1..20].

List of arithmetic utterances for TIMES and OVER	
201. ***Zero TIMES Zero***	301. ***Zero OVER Zero***
202. ***Zero TIMES One***	302. ***Zero OVER One***
... **. . .**	... **. . .**
210. ***Zero TIMES Nine***	310. ***Zero OVER Nine***
211. ***One TIMES Zero***	311. ***One OVER Zero***
... **. . .**	... **. . .**
220. ***One TIMES Nine***	320. ***One OVER Nine***
... **. . .**	... **. . .**
... **. . .**	... **. . .**
300. ***Nine TIMES Nine***	400. ***Nine OVER Nine***

Table 3: List of arithmetic utterances (*, /)

Having this organization of the data make it easy to work with the individual folders and files and gives an intuitive view of the whole dataset.

2.2. Data Preparation

Once obtained the crude data we have to prepare them for Kaldi. The first thing to do is to convert the audio files from the original format (Opus) into Kaldi's friendly format.
Kaldi demand that the input audio samples to be in WAVE (.wav) format and to have certain audio features. This features include *channel* that has to be "mono" and *sample rate* included in the set {8 KHz, 16 KHz, 32 KHz}. If you fail to provide this characteristics, Kladi will complain and would not be able to execute the process. In this context the best thing to do is to convert the audio samples in the acceptable format. There are many tools that can achieve this result and you can choose depending on your own taste. In our case Linux utility *sox* was just the right one. In the case you have trouble using command line tools you can try *Audacity* that has a intuitive GUI interface. Here is the sox instruction to convert fileName.ogg in fileName.wav with the right features.

```
$: sox fileName.ogg -t wav -r 16000 -c 1 - remix 2 fileName.wav
```

Once we have the data in the right format we are ready to go. In the next section we are going to see how to install and configure Kaldi, creating in this way the execution environment. We are going to follow a step-by-step approach because Kaldi is an expert oriented tool and sometimes results difficult to compile and resolve the software dependencies. Also is important to know that Kaldi can be seen as a conglomerate of scripts and C++ source files (.cpp).

3. Kaldi

Kaldi is the software that we are going to use to train and test our data. Kladi comes under a Apache licence and is written in C++. This make possible the extension and the low level interaction with the system. On the other hand Kaldi makes an extensive use of broadly available open source software and libraries. Is important to mention that Kaldi is suitable to be installed and used on a Linux environment. In our case we are going to install Kaldi on a GNU/Linux system, precisely "Ubuntu 18.04.1 LTS". The machine contains also a quad core "Intel(R) Core(TM) i5-2520M @ 2.50GHz" processor and 4 GB of RAM.

3.1. Installation

Kaldi is maintained as a GitHub repository. This imply that before installing Kaldi is better install *git*. This can be easy done executing the following command on the terminal.

```
$: sudo apt-get install git
```

In addition to *git*, Kaldi need the following list of software:

- *atlas* or *blast* - calculations in the field of linear algebra.
- *autoconf* - software that helps in the compiling process
- *automake* - create the Makefile
- *libtool* - creating static and dynamic libraries
- *wget* - use by Kaldi to retrieve files on the net.
- *awk* - programming language that deals with pattern processing
- *grep* - system utility for text searching
- *make* - to build the source code

- *perl* - programming language, ideal for expression manipulation

The great news is that the majority of this packages comes with the operating system itself, but is better to check and update them before compiling Kaldi. Having put together all the pieces now is time to download and install Kaldi. We are going to download Kaldi in our *Home directory*. The instructions to do this are the following:

```
$: git clone https://github.com/kaldi-asr/kaldi.git kaldi –origin upstream
$: git pull
```

Once finished the download we can explore inside Kaldi directory. The text file *INSTALL* is our guide to the installation process. Here are the instruction to execute in the specified order:

```
$: cd kaldi/tools
$: extras/check_dependencies.sh
$: make
$: cd kaldi/src
$: ./configure
$: make depend
$: make
```

Note that the compiling process can take a long time to end (in our case it took roughly 45 minutes). In case you want to speed up the process you can insert the *-j* option in the make command. For example *"make -j 4"*, where 4 is the number of processors. If the compiling process goes smoothly then Kaldi will be up and ready to run. In any case is a good idea to test the installation before employing the system. Just run the instructions on your terminal.

```
$: make test
$: configure
```

Now we are ready to explore Kaldi's structure and organization.

3.2. Overview

Kaldi is a sophisticated ASR system that contains almost all the algorithms used in acoustic model training. Kaldi provides also a very rich set of ready to use models. All this models are contained in the *"egs"* directory. Some of the available models are *"wsj"* for the Wall Street Journal Corpus, *"timit"* for TIMIT and *"rm"* for Resource Management Corpora. The *"egs"* folder is the place where we are going to create our own project directories. Another important folder in the first level of Kaldi's hierarchy is *"tools"* that contains all the external files and libraries needed for the normal function of the system. The third important folder is *"utils"*. It contains all the scripts used in the phases data preparation and training. The figure 2 shows the structure of Kaldi.

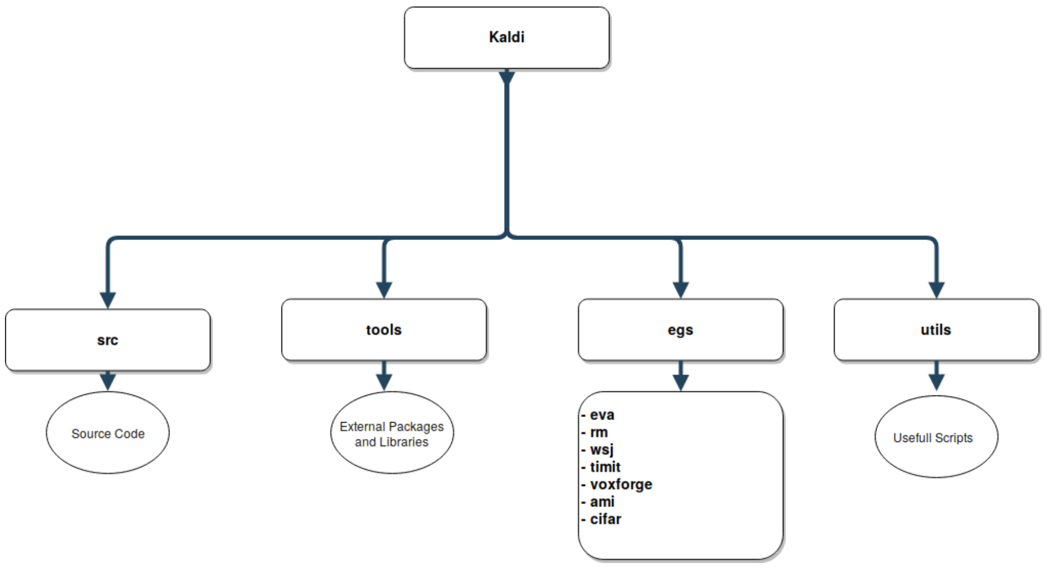

Figure 2: Kaldi's hierarchy

In the root directory there are of course other folders and files but we intentionally mentioned the once that we are going to use in our project. At this point the development environment is set up and we are ready to start the project.

3.3. Training Overview

What is training all about?!

We are going to ask this question using a high level approach, at least for the moment. Here is the list of the steps to undertake:

1. **Obtain the acoustic data**
   Is the process of creation of the actual audio samples. We have described extensively this process in the second section.
2. **Produce a transcript of the data**
   We also need to transcript the utterances. This consists of producing a text file containing a sentence for each corresponding utterance. This transcription will be used agin in the phase of performance evaluation.
3. **Create the input files for Kaldi in the specified format**
   In this step we are going to create all the input files Kladi needs in order to be able to create an acoustic model. The files that we have to produce have a concise and specified format. In the next section we are going to describe the actual process of creation of this files.
4. **Extract the acoustic features form the audio samples**
   The first step in any training process is the extraction of the acoustic features of the audio samples. Kaldi uses *Mel Frequency Cepstral Coefficients (MFCC)* and *Perceptual Linear Prediction (PLP)* as the basic features. The way this data is computed exceeds the scope of this paper but the underlying idea is based on the linear transformation of the acoustic power spectrum. MFCC and PLP are computed at the earliest phases of the training and act as the basic blocks from which to derive the acoustic model.
5. **Train the monophones and triphone models**
   In this project we are going to perform a *MONO*, (monophone) and a *TRI1* (simple triphone) training. In the case of MONO the training process of the phones does not include contextual information. In other word we do not explore the phones around the phone under examination. To treat monophone training *Gaussian Mixture Model (GMM)* and *Hidden Markov Model (HMM)* are very handy. Kaldi permits the use of *Deep Neural Networks (DNN)* if needed. TRI1 training includes the contextual information, meaning that the left and the right phones surrounding the phone under exam have to be taken in consideration too. This fact lead to a big set of possible triphone from which we have to extract only a small subset. For TRI1, *delta+delta-delta* training, *Linear Discriminant Analysis Maximum Likelihood Linear Transform (LDA-MLLT)* and *Speaker Adaptive Training (SAT)* are the prefered algorithms.

6. **Align audio with the acoustic model** After extracting the acoustic features and training the model a further step of data alignment and optimization can be applied. In this phase very sophisticated algorithms and smoothing procedures are utilized to further improve the model. *Viterbi* algorithm, *Forward-Backward* and *Expectation Minimization* are some of the most used algorithms for the monophones. *Feature Space Maximum Likelihood Linear Regression (fMLLR)* is used as the preferd algorithm for TRI1 alignment and optimization. In-deep description of this algorithms can be found in [3]

3.4. Training

In one of the he's lectures, Dan Povey[7] show the Bayesian formula related to the ASRs. Here is how it look like:

$$P(S|audio) = \frac{p(audio|S)P(S)}{p(audio)} \tag{3}$$

- **S** is the sequence of words (utterance),
- **P(S)** is the language model, n-grams for example,
- **p(audio|S)**is a sentence-dependent statistical model of audio production trained from data,
- **p(audio)** is used for normalization purposes and is not relevant in the process.

Our goal is to create an acoustic model that given a test utterance, choose **S** in order to maximize **P(S|audio)**, the most likely sentence.
In the rest of the section we are going to describe the actual project development step-by-step.
First we have to enter the *"egs"* directory and create a new folder with the name of the project.

```
$: cd kaldi/egs
$: mkdir eva
```

[7]http://www.danielpovey.com/files/Lecture1.pdf

Once created the new directory has to be enriched with some data. First we need access to *"utils"*, *"steps"* and *"src"* and *"path.sh"* script. In order to save some space we are going to link the directories and to copy the file from *"wsj"*.

```
$: cd eva
$: ln -s ../wsj/s5/steps
$: ln -s ../wsj/s5/utils
$: ln -s ../../src
$: cp ../wsj/s5/path.sh
```

At this point we have to correct the *"path.sh"* as follows:

```
$: vim path.sh
$: export KALDI_ROOT='pwd'/../../..
$: export KALDI_ROOT='pwd'/../..
```

Now we are going to create the *"myaudio"* folder that will contain our audio files. Inside this folder we are going to create two new folder, one called *"train"* that will contains the files that will be used to train the model and the second called *"test"* to test it.

```
$: cd egs
$: mkdir myaudio
$: cd myaudio
$: mkdir train
$: mkdir test
```

In *"egs"* we have to create another working folder that we call *"data"*. Inside data we have to create three more folders, *"train"*, *"test"* and *"local"*.

```
$: cd egs
$: mkdir data
$: cd data
$: mkdir train
$: mkdir test
$: mkdir local
```

At this point we have to create four text files in Kaldi's format. This files are going to populate *"train"* and *"test"* folders and will represent the acoustic data. The contents of each file is related to the folder it reside.

- **spk2gender**: contain a list of pairs <*speakerID speakerGender*> for all the speakers in the train folder.
  Here it is an extract of that file:

  ```
  dc m
  ea m
  eu m
  js f
  rk f
  ...
  ```

- **wav.scp**: connects each utterance with the exact audio file. It is a list of pairs that follows the pattern: <*utteranceID pathFile*>.
  Is very important to list all the files present in the train folder, one for each line.

- **text**: as implied by the name this file contain the transcription of each utterance. The pattern is: <*utteranceID utteranceTranscription*>.
  Here it is an extract of the text file for the general utterances. Even here is extremely important to list all the utterances, one at each line.

  ```
  dc2018_26122018_M_001 HELLO EDWARD
  dc2018_26122018_M_002 WHAT IS YOUR NAME
  dc2018_26122018_M_003 WHAT TIME IS NOW
  dc2018_26122018_M_004 CALL MY WIFE
  dc2018_26122018_M_005 SWITCH ON THE LIGHT
  dc2018_26122018_M_006 READ MY BOOK
  dc2018_26122018_M_007 PLAY SOME MUSIC
  ...
  ```

- **utt2spk**: this file connect the utterances with the speakers following the pattern, <*utteranceID speakerID*>.

```
ea2018_27122018_M_001 ea
ea2018_27122018_M_002 ea
ea2018_27122018_M_003 ea
ea2018_27122018_M_004 ea
ea2018_27122018_M_005 ea
ea2018_27122018_M_006 ea
...
```

This files have to be created inside the *"test"* folder also with the corresponding data.
Inside the *"local"* folder we have to create five files that will represent the language data.

- **corpus.txt**: contains the transcription of all the utterances. This file contains 200 lines for the general utterances and 800 for the arithmetical operations.

- **lexicon.txt**: is one of the most important files in the whole project and the one that is the most susceptible. It contains the phonetic transcription of all the words used in the dataset. The best thing to do is to find a good phonetic dictionary for the English language. We used *The CMU Pronouncing Dictionary*[8].
  Here is an extract of *lexicon.txt*.

```
SIL SIL
A AH
AGENDA AH JH EH N D AH
AN AE N
BIRTHDAY B ER TH D EY
BOOK B UH K
CALL K AO L
CHECK CH EH K
DIM D IH M
EDWARD EH D W ER D
...
```

[8]http://www.speech.cs.cmu.edu/cgi-bin/cmudict

The pattern that *"lexicon.txt"* follows is < *word phone1 phone2...* >. *SIL* is a special character that is used to represent silent phones.

- **nonsilence_phones.txt**: contains the list of non silent phones.

```
AE
AH
AO
AW
AY
B
CH
D
...
```

In our general utterances dataset there are 34 phones placed in 34 lines in the file. In the case of arithmetic operations dataset there are only 21 phones. Is important to notice that the phonemes are ordered in alphabetic order.

- **silence_phones.txt**: represent the set of silent phones. This file contains only *SIL* in our project.

- **optional_silence.txt**: contains other optional silent phone. In our case contains *SIL*.

We have almost reach the final stage of our project setting.
One thing that we need to be sure before proceeding further is the presence of a language modelling toolkit. SRILM[9] is the best choice for this task. Usually the package is provided by Kaldi, if not you have to download and install it. For additional information about SRILM consult Stolcke [4].
One of the last thing to do before executing the project is to create the configuration files. Here is how we are going to proceed. First we create a new folder called *"conf"* inside the first level of the directory hierarchy of our project. Once created the folder we have to populate it with two files, namely *"decode.config"* and *"mfcc.conf"*. The first file contain this three lines of code:

[9] `http://www.speech.sri.com/projects/srilm/download.html`

```
first_beam=10.0
beam=13.0
lattice_beam=6.0
```

The second file contains two lines of code:

```
–use-energy=false
–sample-frequency=8000
```

In order to be able to run the project we need to fix the *"path.sh"* script that contains all the links to the files needed for the execution and to create three new files on the top level of the projects directory. Here is the last four steps...
We have to create the *"cmd.sh"* script that is used to tell Kaldi how to execute the program. In our case it has to be in the local CPU without external cluster.

- **cmd.sh**: contains two lines of code and forces Kaldi to execute *"run.pl"* script.

  ```
  export train_cmd=run.pl
  export decode_cmd=run.pl
  ```

- **path.sh**: contain the path settings. We have to be careful here to include everything needed for the project, otherwise Kaldi will complain.
- **run.sh**: a long script that is used to automate the whole executing process. Instead of running the training commands one-by-one run all together in a row.
- **clean.sh**: an utility script that clean up all the files created when the project is executed.

Finally everything is set up and we are ready to run the project and hopefully to get some results.

3.5. Execution

The execution at this point is as easy as to run a simple script, *"run.sh"*. The process will take some time based on the way we have set the data for the different experiments. At the end of each execution we will be able to see the performance results. The commands below are used to run the program.

```
$: cd eva
$:../run.sh
```

Here is how the results look like:

```
%WER 20.00 [ 18 / 90, 0 ins, 7 del, 11 sub ]
%SER 40.00 [ 12 / 30 ]
exp/tri1/decode/wer_11
%WER 20.00 [ 18 / 90, 0 ins, 7 del, 11 sub ]
%SER 40.00 [ 12 / 30 ]
exp/tri1/decode/wer_12
%WER 20.00 [ 18 / 90, 0 ins, 7 del, 11 sub ]
%SER 40.00 [ 12 / 30 ]
. . .
```

In the next section we are going to perform a set of experiments with the goal to evaluate the performance of the system in different setups.

4. System Evaluation

The evaluation of the system is the last step in our development. Knowing that the size of the dataset and the quality of the data are of extremely importance in the training phase we were eager to achieve the best possible results. We are going to evaluate the performance going thorough a series of experiments with various arrangements of the test and train data. At the end of each experiment the performance data will be shown. There are two ways how the results of the elaboration can be viewed. The first one is to search inside *"mono/decode"* and *"tri1/decode"* directories for *wer_number* files. The second approach consists of creating a script file that mechanizes this process. We opted for the last choice.

4.1. Performance Metrics

In the filed of *speech recognition* and *machine translation* there is a well accepted set of metrics used to certify to accuracy of a system. In our experiments we are going to use *Word-Error-Rate* **(WER)** and *Sentence-Error-Rate* **(SER)**. The formula below show how to compute **WER**.

$$WER = \frac{S + D + I}{S + D + C} \tag{4}$$

- S is the number of words that have been substituted
- D is the number of deleted words
- I is the number of inserted words
- C is the number of correct words

SER is evaluated in the same fashion but instead of words we consider whole sentences.

4.2. Experiment 1 - miniEva

The goal of this experiment is to reveal the system accuracy in a dataset that consist of 60 utterances form the general utterance set. We have divide the data in 1-3 ratio. The audio samples in the test set are a mixture of utterances, one from each participant. The result after the execution are shown in Table 4.

Performance results for miniEva			
Train set	***Test set***	***WER %***	***SER %***
40 utt.	20 utt.	90%	100 %

Table 4: Exp. 1 results

This data shown that in this setup the system was practically unable to recognize the input producing a very bad results. We believe the catastrophic outputs are provoked by the multiplicity of the participants in the test and train set. The system was not able to purely identify the exact features to use in order to recognize the utterances leading to an erroneous acoustic model. This fact can be observed in the enormous number of deletions and substitutions in the *mono* and *tri1* phases.

4.3. Experiment 2 - Eva

In this experiment we are going to increase the number of data in the train set hoping to get better results than those of the previous experiment. In the new setup the train set contain 160 utterances from eight different participants and the test set consists of 40 utterances from two participants.

Performance results for Eva			
Train set	***Test set***	***WER %***	***SER %***
160 utt.	40 utt.	95 %	95 %

Table 5: Exp. 2 results

Even though the resulting data are disappointing, we observe some new features emerging.
The first one is related with a slightly decrease on the **WER** and **SER** values. This small quantity of improvement do not justify the quadruple size of the train set.
The second thing that we observe is an apparently strange increase of the number of *insertions*. In the first experiment this value was zero or less than 3, here instead this value dominate the components of **WER** and **SER**. The data convinced us that increasing the number of distinct participants does not improve the accuracy of the system.
Given the fact that the training process was fast (around 2 minutes for this instance) we experimented different types of data arrangements. The second row of the previous table show a setup where the train set had 120 utterances and the test set had 80 utterances. The result were even worst. In the new next experiments we are going to consider the *Arithmetical operations* dataset.

4.4. Experiment 3 - Cal

In this group of experiments we are going to use the second dataset. The characteristic of this dataset is that it consists of only two participants but the number of utterances for each of them is bigger. In the first setup we are using the utterances of one participant. We have 100 utterances of type *number PLUS number*, from which 10 we use for test and 90 for train.
The results were unexpected considering the previous attempts. We have 0 % **WER** and 0 % **SER**. The accuracy data that the system output are of course biased by the fact that the same participant is used for the training

and for the testing. In order to reinforce this conclusion we added 10 more utterances performed by a second participant in the test set. The result are shown in the second row of Table 6.
As you can see in the Table with the increase of *entropy* in data the **WER** and **SER** values increase and this is what we expected.
The values of **WER** and **SER** are the same in both *mono* and *tri1* training, but the only thing that changes are their components. In the *mono* training we observe all the components, *insertion, deletion* and *substitution*, on the hand in *tri1* training *deletion* is missing.
In our third setup we enriched the test set with one more participant. The results were as expected, the **WER** and *SER* values increased as shown in Table 6. Notice that in all three setups the train set is kept static.

Performance results for Cal			
Train set	***Test set***	***WER %***	***SER %***
90 utt.	10 utt.	0 %	0 %
90 utt.	20 utt.	6.67 %	20 %
90 utt.	30 utt.	15 %	37 %

Table 6: Exp. 3 results

To recap the observations of experiment 3 were in line with our expectations. The system accuracy decrease with the augmentation of the new data in the test set.

4.5. Experiment 4 - MaxCal

In this last experiment we are going to use a much larger dataset that contains 800 utterances performed by two participants and a small test set containing at maximum 30 utterances. Unfortunately, even though we prepare very well the experiment, we were not able to improve the accuracy. In this test we obtained 30 % accuracy. Table 7 show the data.

Performance results for MaxCal			
Train set	***Test set***	***WER %***	***SER %***
800 utt.	10 utt.	33 %	100 y %

Table 7: Exp. 4 results

4.6. Now what?!

- How we are going to use the model that we have created?

This is very natural question that arise after all this work that we have done. In order to create a speech recognition system we need to find a way to decode to input audio. We need a *transcription* of the most probable sentence given the utterance. Keep in mind that the main reason we use Kaldi is to create an acoustic model. How we use the model Kaldi creates is up to our needs and goals. The best way to obtain a text file containing the transcription of the input audio is to proceed as follow:

INPUT:
1. *transcriptions/wav.scp*
2. *conf/mfcc.conf*
3. *experiment/tri1/final.mdl*
4. *experiment/graph/HCLG.fst*
5. *experiment/graph/words.txt*

We have to create a new folder inside *maxcal* called *transcriptions*. This new folder contain one file, **wav.scp**. The format of **wav.scp** is the standard one we have seen previously. This file is related with the utterance we want to transcript. The other files that we use as input for the transcription are produced during the training phase and represent the model created by Kaldi. Detailed description of this files are provided in the Kladi's website.

Once completed this preparatory step we are ready to start the decoding procedure by lunching *decode.sh* script. To analyze the output of this process we have to check the interior of *transcriptions* folder. A first glance inside this folder reveal six new files. At this point we are interested on the only human readable file, *one-best-hypothesis.txt*. This file contain the transcription of the input utterance. Based on the quality of the created model we shall see different levels of accuracy in the transcription file. In our experiments the level of accuracy changed over and over based on many variables such as the train and test sets but even related to the number of different participants. Typically and increase number of participants leads to a decrease of accuracy. Here is shown the output of the decoding process:

OUTPUT:
1. *transcriptions/wav.scp*
2. *transcriptions/delta-feats.ark*
3. *transcriptions/feats.ark*
4. *transcriptions/feats.scp*
5. *transcriptions/lattice.ark*
6. *transcriptions/one-best.tra*
7. **transcriptions/one-best-hypothesis.txt**

5. Conclusion

At this point is important to say some final words about what we have done in this project.
First, we have described in *"tutorial-like"* fashion the path that we follow in order to create a very small speech recognition system. You will find in this paper a step by step approach related to the data preparation, training and testing of an acoustic model. There is an extensive description of Kaldi and it's use also.
Second, We extensively evaluated the model in order to establish it's performance and accuracy. The results varied based on different parameters that we clearly underlined in our discussion.
Third, we showed the best way to get a transcription of an input utterance. Future work must be done in order to improve the model and it's accuracy by performing more experiments on different parallel machines.

References

[1] D. P. et al., http://kaldi-asr.org/ (2017).

[2] G. Kirby, http://www.geoffkirby.co.uk/zlpfslaw.pdf (1985).

[3] D. Jurafsky, J. Martin, Speech and Language Processing, Prentice Hall, 2th edition, 2008.

[4] A. Stolcke, Srilm an extensible language modeling toolkit (2002).