Speech Processing

Final Project

University of Trento

Human Language Technology and Interfaces

By: Seid Muhi Yimam

Submitted To: **Marco Matassoni**

January 28, 2011

Acknowledgement

First and for most, I would like to thank the lab course professor, **Marco Matassoni**, for his continuous replies and explanations for my request. I appreciate him for the timely and details responses he provided by e-mail. I also would like to thank the course professors, **Giuseppe Riccardi** and **Fabio Brugnara** for their detailed and clear lecture presentations. I would also thank my classmates for their cooperation. Thank you all.

1. **Given the sequences of Items soled by the person, we are to search most probable cities the person has sold those items. Here, the idea is to find maximum probabilities of cities where those items were sold. We have Cities A, B, and C. the sequences of Items sold are T F T T W W F W F D D D D D T W T D W F D W W F D F D T T D.**

The salesman problem can be solved with the Viterbi Algorithm. Viterbi Algorithm is a special HMM problem where the task is to find the most probable hidden HMM states from the observed sequences. In the sales man problem, the hidden states are the cities where the items are sold.

Now let's tackle the problem from the Viterbi algorithm point of view.

The initial probability, $\mathbf{pi_i}$={1,0.0,0.0} where i={A,B,C}. We have been told that he has started from City A and hence it is 0% probable to be at City B or C.

The state transition probability ($\mathbf{a_{ij}}$), the probability to move to the next city i from city j, is given in the problem statement. Table 1.1 shows the state transition probability table.

Table 1.1: state transition probability of the salesman problem

	Next Village		
Current Village	A	B	C
A	41%	24%	35%
B	27%	39%	34%
C	33%	44%	23%

Similarly, the emission probability ($\mathbf{b_{ik}}$), the probability of selling an Item k while in city i is also explicitly stated in the problem statement. Table 1.2 shows the emission (confusion) probability table.

Table 1.2: the emission probability for the salesman problem

	Item			
Village	T	F	W	D
A	36%	15%	29%	20%
B	23%	21%	13%	33%
C	21%	27%	38%	14%

Finally, the observation sequences as stated in the problem are:

S= T F T T W W F W F D D D D D T W T D W F D W W F D F D T T D

Then, as per the Viterbi algorithm formulations, our objective is to found out the most probable sequences of hidden states (X) that might generate the observed sequences (S).

Below is the summary of the Viterbi Algorithm initialization, recursion, output and backtrack algorithm as implemented with java programming language.

The items sold are accepted from an input box (TextField) and the number of Items sold, that is T,F,D,… will be calculated and character tokenizer is used to chop up the input characters (T,F…) into array of characters. Then two dimensional arrays of size 3 by number of total items sold is created to store the best path value and back pointer at each time instances and for each states. Two dimensional arrays are also created to store the state transition probability and the emission probability. The initial best probabilities are also calculated from initial probabilities and emission probabilities.Following are the java code excerpt for the above explanations

```
//observation from the user changed to upper case and stored as array of characters
char[] ObsStates=txtObservedSeq.getText().trim().toUpperCase().toCharArray();
int len=0;
        for (char c: ObsStates){
                len++;
                ItemSold.add(c);
        }
//emmision and transition probabilities
        double a[][]={{0.41,0.24,0.35},{0.27,0.39,0.34},{0.33,0.44,0.23}};
                double b[][]={{0.36,0.15,0.29,0.20},{0.23,0.21,0.13,0.33},{0.21,0.27,0.38,0.14}};
                //initial probability. A has 100% probability
                double pi[]={1.0,0.0,0.0};
                //the best path value at each time for each state
                double VT[][]=new double[3][len];//most probable path
                //the best path at each time for each state
                int BP[][]=new int[3][len];//back pointer
                BP[0][0]=BP[1][0]=BP[2][0]=0;//Initial Backpointer
                int BPF=0;//final backpointer to the best final prob value
                //Intial Probabilities best paths
                VT[0][0]=b[0][0]*pi[0];
```

```
VT[1][0]=b[1][0]*pi[1];
VT[2][0]=b[2][0]*pi[2];
```

Then, the java for loop is used to incrementally calculate the best path of each state at each time and the best back pointer for that state to find the best previous path. To calculate the best path, First the Type of item sold is determined from the array character at that particular time. Then the maximum of the multiplications of the best previous paths with each state transition probability is determined and multiplied again by Items emission probability for different states. Similarly, for each state's best path, its best previous best path is retained by calculating the previous best paths for each state. The following java cod snippet shows how the best paths and back pointers are calculated.

```
for(i=0;i<len;i++){
        String ob=ItemIterator.next().toString();
        if(firstcity){
            firstcity=false;
        }else{
            //System.out.println(ob);
        if(ob.equals("T")){
            VT[0][i]=max(VT[0][i-1]*a[0][0],VT[1][i-1]*a[1][0],VT[2][i-1]*a[2][0])*b[0][0];
            BP[0][i]=BPt(VT[0][i-1]*a[0][0],VT[1][i-1]*a[1][0],VT[2][i-1]*a[2][0]);
            VT[1][i]=max(VT[0][i-1]*a[0][1],VT[1][i-1]*a[1][1],VT[2][i-1]*a[2][1])*b[1][0];
            BP[1][i]=BPt(VT[0][i-1]*a[0][1],VT[1][i-1]*a[1][1],VT[2][i-1]*a[2][1]);
            VT[2][i]=max(VT[0][i-1]*a[0][2],VT[1][i-1]*a[1][2],VT[2][i-1]*a[2][2])*b[2][0];
            BP[2][i]=BPt(VT[0][i-1]*a[0][2],VT[1][i-1]*a[1][2],VT[2][i-1]*a[2][2]);


            }
…
```

For the last item, the final best path and its state is determined. Once its best path is determined, hence final best state, the best previous paths are navigated back till the first hidden state. The best hidden states tracked are stored in the array and its content is displayed from last to first (n-1 to 0 index of the array) to display its content from the first hidden state to the last. The code snippet below shows this scenario.

```
String[]Bestpaths=new String[len];
```

```
int bestp=0;//best path for the last state
    if(max(VT[0][len-1],VT[1][len-1],VT[2][len-1])==VT[1][len-1])
            bestp=1;
    else if (max(VT[0][len-1],VT[1][len-1],VT[2][len-1])==VT[2][len-1])
            bestp=2;
    //System.out.println(VT[0][len-1]+" "+VT[1][len-1]+" "+VT[2][len-1]+" best State "+bestp);
    if(bestp==0)
            Bestpaths[len-1]="A";
    else if(bestp==1)Bestpaths[len-1]="B";
    else Bestpaths[len-1]="C";
    for (i=len-1;i>0;i--){
            bestp=BP[bestp][i];//backtracking
            if(bestp==0)Bestpaths[i-1]="A";
            else if(bestp==1)Bestpaths[i-1]="B";
            else Bestpaths[i-1]="C";
    }
    Bestpaths[0]="A";
    StringBuilder sb=new StringBuilder();
    for(i=0;i<len;i++){
            sb.append(Bestpaths[i].toString());
    }
            txthiddenstate.setText(sb.toString());//display on the output TextField
    }
```

The full java code (**SP2.java**) is attached with the resources package. The final output (ACAAACBCBBBBBBACBBCBBCACB CBAAA), with a graphical representation is given in figure 1.

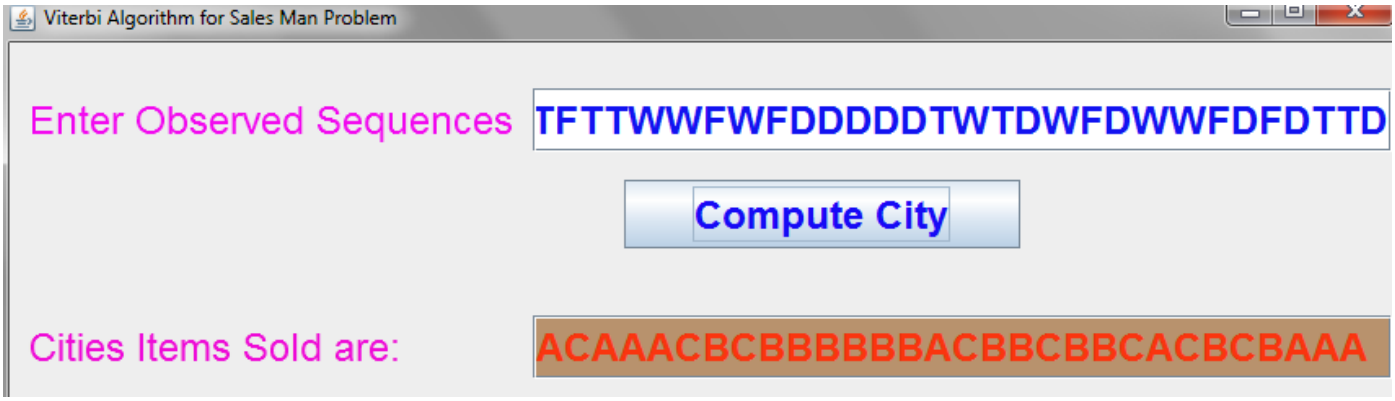

Figure 1.1: The cities the salesman soled the selected items.

2. Considering the signals in folder ex2/timit:

2.1. compute the average pitch of each sentence;

Pitch represents the perceived fundamental frequency of a sound [1]. The pitch is very important characteristics of a sound such as in identifying whether a sound is accented or not, voiced or unvoiced and son. From the timit folder, the different sounds stored are analyzed. The detail implementation of the average pitch is as follows.

Praat scrip is written to read sound file(s) in a directory and calculate the mean pitch of each sentence based on praat's built-in pitch mean calculator. As the sound files are arranged in hierarchical directories, we have used the Java programming language environmental **system exec** capability to call the praat scrip and fed the unique directories to it. Then the praat scrip will execute each sound files in a hierarchy, compute the mean pitch and append the result to a files system. The praat script and the java program are shown as follows.

Praat script - **getmeanpitch.praat**

```
#this form will get its input argument from the java program
form Get file and compute mean pitch
    sentence Source_directory
endform
#creates list of files as list of strings from a given directory, as some directory have more than one files
Create Strings as file list... list 'source_directory$'/*
```

```
#total number of sound files in the directory
file_count = Get number of strings
for current_file from 1 to file_count
  select Strings list
  filename$ = Get string... current_file
  #get the sound files
  Read from file... 'source_directory$'/'filename$'
  #pitch countour in 0-600 ranges,
 To Pitch... 0 75 600
 #Get the mean pitch (0.0 0.0 - all avalable pitch values will be considered)
 meanp = Get mean... 0.0 0.0 Hertz
 #write the output a file, appending each time the mean value with its name
fileappend "results.txt" 'source_directory$'/'filename$' 'meanp"newline$'
 echo 'meanp' //we use it from the java program to determine number of female and males spks
endfor
```

Java program - **GetMeanPitch.java**

```
List<File> files = GetMeanPitch.getFileListing(new File("/home/abuyusra/Desktop/test");
//gives directory name to praat, where the java runtime opens the praat to execute
   for(File file : files ){
         //praat system call, the praat script, and its argument directory
         String cmd = "praat /home/abuyusra/"+"getmeanpitch.praat "+file;
         Runtime run = Runtime.getRuntime();
         Process pr = run.exec(cmd);
         pr.waitFor();
         BufferedReader buf = new BufferedReader(new InputStreamReader(pr.getInputStream()));
                  String line = "";
                  while ((line=buf.readLine())!=null) {
                           System.out.println(line);
         }
   }
```

The function **getFileListing** is included in the full java file in the resource directory (**GetMeanPitch.java**). The output is a file system (**Praat_results.txt**, in the resource folder) which contains the mean pitch of all files with their full path name. Based on the analysis of pitch ranges, (70-250 Hz for men, 150-400 Hz female, and 200-600 Hz kids) [2] we have observed that

- Male pitches are less than **189 Hz**
- Female pitches are greater than **162 Hz**
- Same files (same sentences) do have different mean pitch for different same sax speakers (may be due to differences in age and other characteristics).

By rough estimates (considering male's mean pitch to be exclusively below 150, females mean pitch to be above 200, and considering the value in between to be either male or female, we have got female speaker to be 21, male speaker 53 and both sex 26.

2.2. build a word-loop grammar from the list ex2/wordlist;

To build the word loop grammar, we will use the htk command

HBuild wordlist wordLoop

Where **wordList** is list of all words used for the acoustic model in the absence of Language model to predict the most likely sequence of words [3]. The command generates a word loop grammar, which is used to put all words of the vocabulary in a loop and therefore allows any word to follow any other word. The word loop grammar simply contains a network in which each word occurs with equal probability. The **wordLoop** file is included in the resource folder as a reference. The word loop grammar generated will be used for decoding in the next procedure.

2.3. With the resulting wordnet, decode the signals and evaluate the Word Error Rate creating the proper reference MLF from ex2/timit.txt; the lexicon is ex2/5k.lex, the acoustic models are in folder ex2/hmm/.

HTK provides a recognition tool called HVite which uses the token passing algorithm to perform Viterbi-based speech recognition. The ex2 ex2/hmm folder contains the training HMMs and all the test sound wave with an encoded mfc files. To run Hvite, the script file which contains path of all mfc files is created using the java program (**CreateMfcScp.java**) as **test.scp** script file. Once the file is created we do run the Hvite command changing its -p parameter to improve performance.

HVite -H hmm/hmmdefs -C config -S test.scp -i out.mlf -w wordLoop 5k.lex phones

In this command, **hmmdefs** is the HMM that is created during training, **config** contains the configuration file, **test.scp** is a script file which contains the path of each mfc files, **out.mlf** is the mlf file that we will use for computing word error rate, **wordLoop** is the word loop grammar created in 2.1 above, **5k.lex** and **phones** are the lexicon and all available phones.

Once the Hvite command generates the mlf files, we have to run the HResults tool to evaluate its performance. The mlf transcription file (reference label file) for the test data is generated from the **timit.txt** file using java program (**FormatToMLF.java**) and the **prompts2mlf.pl** Perl script from the HTK resource. The java program make ready the **timit.txt** file in the correct format that the **prompts2mlf.pl** script needs such as removing the start and end times, removing the txt file name extension and so on. Based on the mlf file output by Hvite and the reference transcription mlf file generated by prompts2mlf.pl, HResults is executed as follows:

HResults -I words.mlf phones out.mlf

After tuning **-p** parameter of Hvite for different values (0.1 is the best parameter expermented), we have found the following final output of HResults.

```
====================== HTK Results Analysis =======================
  Date: Fri Jan 28 12:23:09 2011
  Ref : words2.mlf
  Rec : out1.mlf
------------------------ Overall Results --------------------------
SENT: %Correct=0.00 [H=0, S=100, N=100]
WORD: %Corr=11.17, Acc=-75.40 [H=84, D=9, S=659, I=651, N=752]
============================================================
```

Observation: the HResults performance analysis shows that, no sentences is recognized correctly. This is due to the word loop grammar is developed from free word lists. Similarly only 84 word alignments are found correct while 9, 659, and 651 errors are encountered due to deletion, substitution and Insertions respectively.

3. Build n-gram language models from the corpus ex3/holmes.txt and evaluate the resulting perplexities on the sets ex3/hislastbow.txt, ex3/lostworld.txt, and ex3/otherauthors.txt. The parameters of the n-gram models should be evaluated for different settings and the perplexity results discussed.

A language model is usually formulated as a probability distribution **p(s)** over strings **s** that attempts to reflect how frequently a string **s** occurs as a sentence [6].

For this problem, SRILM toolkit is used. SRILM is a collection of C++ libraries, executable programs, and helper scripts designed to allow both production of and experimentation with statistical language models for speech recognition and other applications. The main purpose of SRILM is to support language model estimation and evaluation [4]. The basic SRILM modules we used for this project are **ngram-count** and **ngram**

ngram-count generates and manipulates N-gram counts, and estimates N-gram language models from them. The program first builds an internal N-gram count set, either by reading counts from a file, or by scanning text input.

Basic ngram-count commands:

ngram-count -order *N* -text *file.txt* -write *file.cnt*

Writes N-gram count file.cnt from file.txt to a file system.

ngram-count -order *N* -text *file.txt* -lm *file.lm*

Generates a modlel file file.lm from file.txt files and write it to file system.

The second important SRILM model used in this project is **ngram**. **ngram** performs various operations with N-gram-based and related language models, including sentence scoring, perplexity computation, sentences generation, and various types of model interpolation [5].

The **perplexity** of a model is the reciprocal of the (geometric) average probability assigned by the model to each word in the test set [6]. It is a measure of the size of the set of words from which the next word is chosen given that we observe the history of spoken words. The lower the perplexity, the better the unseen sentences fit the corpus underlying the given language model. Below is the detail description of perplexity computations for the three corpuses.

./ngram-count -text ~/SP/ex3/holmes.txt -order 3 -write ~/SP/ex3/holmes.cn3

Counts the tri-grams from the corpus and write out the output to file system.

./ngram-count -text ~/SP/ex3/holmes.txt -order 2 -write ~/SP/ex3/holmes.cn2

Counts bi-grams from the corpus and write out the output to file system.

Then, the model for bigram and trigram is generated using the following commands.

./ngram-count -read ~/SP/ex3/holmes.cn2 -order 2 -lm ~/SP/ex3/holmes.lm2 -gt1min 2 -gt1max 7 -gt2min 2 -gt2max 7 -gt3min 2 -gt3max 7

./ngram-count -read ~/SP/ex3/holmes.cn3 -order 3 -lm ~/SP/ex3/holmes.lm3 -gt1min 3 -gt1max 7 -gt2min 3 -gt2max 7 -gt3min 3 -gt3max 7

The **gtXmin** and **gtXmax** are Good-Turing discounting for n-gram.

Finally the ngram command for perplexity computation for the three different corpuses is executed as follows:

./ngram -ppl ~/SP/ex3/hislastbow.txt -order 2 -lm ~/SP/ex3/holmes.lm2

Result:

file /home/abuyusra/SP/ex3/hislastbow.txt: 7194 sentences, 91144 words, 2239 OOVs

0 zeroprobs, logprob= -206781 ppl= 141.824 ppl1= 211.77

./ngram -ppl ~/SP/ex3/hislastbow.txt -order 3 -lm ~/SP/ex3/holmes.lm3

Result:

file /home/abuyusra/SP/ex3/hislastbow.txt: 7194 sentences, 91144 words, 2239 OOVs

0 zeroprobs, logprob= -204391 ppl= 133.932 ppl1= 199.061

./ngram -ppl ~/SP/ex3/lostworld.txt -order 2 -lm ~/SP/ex3/holmes.lm2

Result:

file /home/abuyusra/SP/ex3/lostworld.txt: 7092 sentences, 89600 words, 3876 OOVs

0 zeroprobs, logprob= -213193 ppl= 198.129 ppl1= 306.885

./ngram -ppl ~/SP/ex3/lostworld.txt -order 3 -lm ~/SP/ex3/holmes.lm3

Result:

file /home/abuyusra/SP/ex3/lostworld.txt: 7092 sentences, 89600 words, 3876 OOVs

0 zeroprobs, logprob= -212152 ppl= 193.077 ppl1= 298.421

./ngram -ppl ~/SP/ex3/otherauthors.txt -order 2 -lm ~/SP/ex3/holmes.lm2

Result:

file /home/abuyusra/SP/ex3/otherauthors.txt: 3806 sentences, 52516 words, 3300 OOVs

0 zeroprobs, logprob= -122591 ppl= 205.149 ppl1= 309.647

./ngram -ppl ~/SP/ex3/otherauthors.txt -order 3 -lm ~/SP/ex3/holmes.lm3

Result:

file /home/abuyusra/SP/ex3/otherauthors.txt: 3806 sentences, 52516 words, 3300 OOVs

0 zeroprobs, logprob= -122379 ppl= 203.274 ppl1= 306.6

Observation: we can observe that the first corpus, hislastbow.txt, has more similarity to the language model corpus, holmes.txt. We can also observe that the the trigram language model less perplexity to the corpuses.

Note: due to lack of time, the corpus is not normalized to remove unwanted characters and a sentence per line format as per the SRILM suggestions. We also computed one discounting techique for the avalaible SRILM discounting thechniques.

References

1 http://en.wikipedia.org/wiki/Pitch_%28music%29

2 Hartmut Traunmüller and Anders Eriksson, The frequency range of the voice fundamental in the speech of male and female adults

3 Steve Young, Gunnar Evermann, et.al., The HTK Book (for HTK Version 3.4)

4 Andreas Stolcke, SRILM —AN EXTENSIBLE LANGUAGE MODELING TOOLKIT

5 http://www-speech.sri.com/projects/srilm/manpages/ngram.1.html

6 Stanley F. Chen, Joshua Goodman, An Empirical Study of Smoothing Techniques for Language Modeling