Hidden Markov Model Application

Jose Millan
jampmil@gmail.com
Graphical Models
Data Mining & Knowledge Management Masters Program
Università degli Studi del Piemonte Orientale 'Amedeo Avogadro'
Alessandria, Italy

1 Introduction

Hidden Markov Models (HMM), are a statistical Markov model, commonly defined as stochastic finite state machines, in which the system being modeled is assumed to be a Markov process with unobserved or hidden states. Nowadays HMMs are commonly used in pattern recognition (speech, handwriting, gesture recognition, part-of-speech tagging, musical score, etc) and fields like computational biology [4] [10].

The present work implements a python-based solution which aims to provide the same functionality as the C++ Implementation of Hidden Markov Model, by Dekang Lin [5], which offers a solution for the most common challenges of the Hidden Markov models: given a model getting a sequence of observations from it; finding the most probable sequence of states of a model given a sequence of observations; and estimating the parameters of a Hidden Markov Model given a set of observed feature vectors.

2 Theoretical Background

The present work implements a solution for different problems related to a Hidden Markov Model. Before introducing the developed solution certain concepts related to this have to be understood. Below the main concepts are briefly explained.

- State: States are atomic events that can transfer from one to another. The state is not directly visible, but the output (emission), dependent on the state, is visible.
- Transition probability: The probability of transitioning from one state to another. The transition probabilities control the way the hidden state at time t is chosen given the hidden state at time $t-1$.
- Emission/Observation: The outcome, at time t of passing through a state.
- Emission probability: The probability of an emission being thrown by a state. The emission probabilities control the distribution of the observed variable at a particular time given the state of the hidden variable at that time. The emission probabilities do not change over time.

With these concepts in mind, and considering the common challenges solved by the solution implemented, as mentioned in Section 1, a description of the algorithms used to solve these challenges is presented below.

2.1 Viterbi Algorithm

The Viterbi algorithm provides an efficient way of finding the most likely state sequence in the maximum a posteriori probability sense of a process assumed to be a finite-state discrete-time Markov process [6]. This algorithm is commonly used in speech recognition, speech synthesis, diarization, keyword spotting, computational linguistics, and bioinformatics [11]. The pseudo-code of this algorithm, based on the Rabiner approach [7], can be found in Algorithm 1.

Algorithm 1: Viterbi Algorithm

Input: O : The observation space
S : The state space
Π : Array of initial probabilities of size K such that π_i stores the probability that $x_1 == s_i$
Y : Sequence of observations of size T such that $y_t == i$ if the observation at time t is o_i
A : Transition matrix of size $K \cdot K$ such that A_{ij} has the probability of transiting from state s_i to $states_j$
B : Emission matrix of size $K \cdot N$ such that B_{ij} has the probability of observing o_j from state s_i

Output: X : The most likely hidden state sequence of size T

```
begin
    foreach state i ∈ {1, 2, ..., K} do
        T1[i, 1] ← π_i · a_{i y_1} ;
        T2[i, 1] ← 0;
    foreach observation i ∈ {2, 3, ..., K} do
        foreach state j ∈ {1, 2, ..., K} do
            T1[j, i] ← b_{j y_1} · max_k (T1[k, i − 1] · a_{kj}) ;
            T2[j, i] ← arg max_k (T1[k, i − 1] · a_{kj});
    z_T ← arg max_k (T1[k, T]);
    x_T ← s_{z_T};
    for i ← {1, 2, ..., K} do
        z_{i−1} ← T2[z_i, i] ;
        x_{i−1} ← s_{z_{i−1}} ;
```

2.2 Forward-Backward Algorithm

The objective of the forward-backward algorithm is to find the values of the state transition probabilities and the output probabilities for each state for which the likelihood of the observed output sequence is maximized [9]. In summary, the algorithm proceeds by making an initial guess of the parameters (which may well be entirely wrong) and then refining it by assessing its worth, and attempting to reduce the errors it provokes when fitted to the given data. In this sense, it is performing a form of gradient descent, looking for a minimum of an error measure [1]. Algorithm 2 shows the pseudo-code for the Forward-Backward algorithm.

Algorithm 2: Forward-Backward Algorithm

Input: O : The observation space
A : Transition matrix of size $K \cdot K$ such that A_{ij} has the probability of transiting from state s_i to $states_j$
B : Emission matrix of size $K \cdot N$ such that B_{ij} has the probability of observing o_j from state s_i

Output: α : The probability of the partial observation sequence until time t, with state S_j at time t
β : The probability of the partial observation sequence after time t, given state S_i at time t

```
begin
    // Forward Stage
    // initialize α
    for 1 <= j <= N do
        α_j(1) = π_j b_j O_1 ;
    // Inductively calculate α
    for 1 <= t <= T − 1 do
        α_j(t + 1) = (Σ_{i=1}^{N} α_i(t) a_{ij}) b_{j O_{t+1}} ;
    // Backward Stage
    // initialize β
    for 1 <= i <= N do
        β_i(T) = 1 ;
    // Inductively calculate β
    for 2 <= t <= T do
        β_i(t − 1) = Σ_{j=1}^{N} a_{ij} b_{j O_t} β_j(t) ;
```

2.3 Baum-Welch Algorithm

The Baum–Welch algorithm is used to find the unknown parameters of a Hidden Markov Model. The algorithm starts with initial probability estimates, then it computes expectations of how often each transition/emission is used and finally re-estimate the probabilities based on those expectations. This process is repeated until convergence [8]. Algorithm 3 presents the pseudo-code for the Baum-Welch algorithm, based on the Rabiner approach [7].

Algorithm 3: Baum-Welch Algorithm

Input: O : The observation space
S : The state space
Π^0 : Array of initial probabilities of size K such that π_i stores the probability that $x_1 == s_i$
A^0 : Initial Transition matrix of size $K \cdot K$.
B^0 : Initial Emission matrix of size $K \cdot N$.

Output: Π : Array of initial probabilities of size K such that π_i stores the probability that $x_1 == s_i$
A : Transition matrix of size $K \cdot K$ such that A_{ij} has the probability of transiting from state s_i to $states_j$
B : Emission matrix of size $K \cdot N$ such that B_{ij} has the probability of observing o_j from state s_i

begin
  for $n \in \{1, 2, \ldots, N\}$ **do**
    calculate α and β using the forward-backward algorithm ;
    $\gamma_t = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N} \alpha_t(j)\beta_t(j)}$;
    $\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{\sum_{1=1}^{N} \sum_{j=1}^{N} \alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}$;
    $a_{ij}^{(n+1)} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$;
    $b_j(k)^{(n+1)} = \frac{\sum_{t=1, O_t = v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$;

3 Implementation

A command line program was implemented using Python 2.7 [3] and the scientific computing package Numpy [2] to solve the challenges mentioned before. In order to create and load a Hidden Markov Model into the program, this model has to be specified in different configuration files, all located in a single folder that represents a working model. The configuration files needed are described below.

- trans file: The trans file represents the transition matrix of a HMM. The file begins with the initial state INIT, and in each line specifies the transition from one state to another along with its probability in the form <origin_state> <destination_state> <prob>. A final state FINAL has to be specified for the program to function properly. An example of this file can be seen in Figure 1 (left).
- emit file: The emit file represents the emission matrix of a HMM. The each line of the file specifies the probability of an emission being thrown by an state in the form <state> <emission> <prob>. An example of this file can be seen in Figure 1 (center).
- input file: This file contains the sequence of observations that are going to be used as input for both the Viterbi and the Baum-Welch algorithm. Each line of the file contains a sequence of observations in the form <O_1> <$O2$> ...<O_n>. An example of this file can be seen in Figure 1 (right).

.trans File

INIT		
INIT	Onset	1
Onset	Onset	0.3
Onset	Mid	0.7
Mid	Mid	0.9
Mid	End	0.1
End	End	0.4
End	FINAL	0.6

.emit File

Onset	C1	0.5
Onset	C2	0.2
Onset	C3	0.3
Mid	C3	0.2
Mid	C4	0.7
Mid	C5	0.1
End	C4	0.1
End	C6	0.5
End	C7	0.4

.input File

```
C3 C4 C4 C3 C4 C4 C5 C4 C4 C4 C7
C1 C4 C4 C4 C7 C6
C3 C1 C3 C3 C4 C4 C4 C6
```

Figure 1: Examples of trans and emit Files

Having defined the configuration files, a solution was designed in order to correctly separate the responsabilities for the program. Figure 2 shows the UML class diagram of the solution, and its components are described below.

- Class `HmmCmd`: This class is the entry point of the program. It manages the interaction with the user by processing the parameters received. It invokes the HMM model according the specified operation to run.
- Class `HmmIO`: The `HmmIO` class is responsible for reading and writing the different files needed for the program, as well for interpreting and decoding each one of the files to give the respective structure of the contained information.
- Class `Hmm`: The `Hmm` class represents a Hidden Markov Model. This class is responsible as well for executing the different algorithms previously presented.

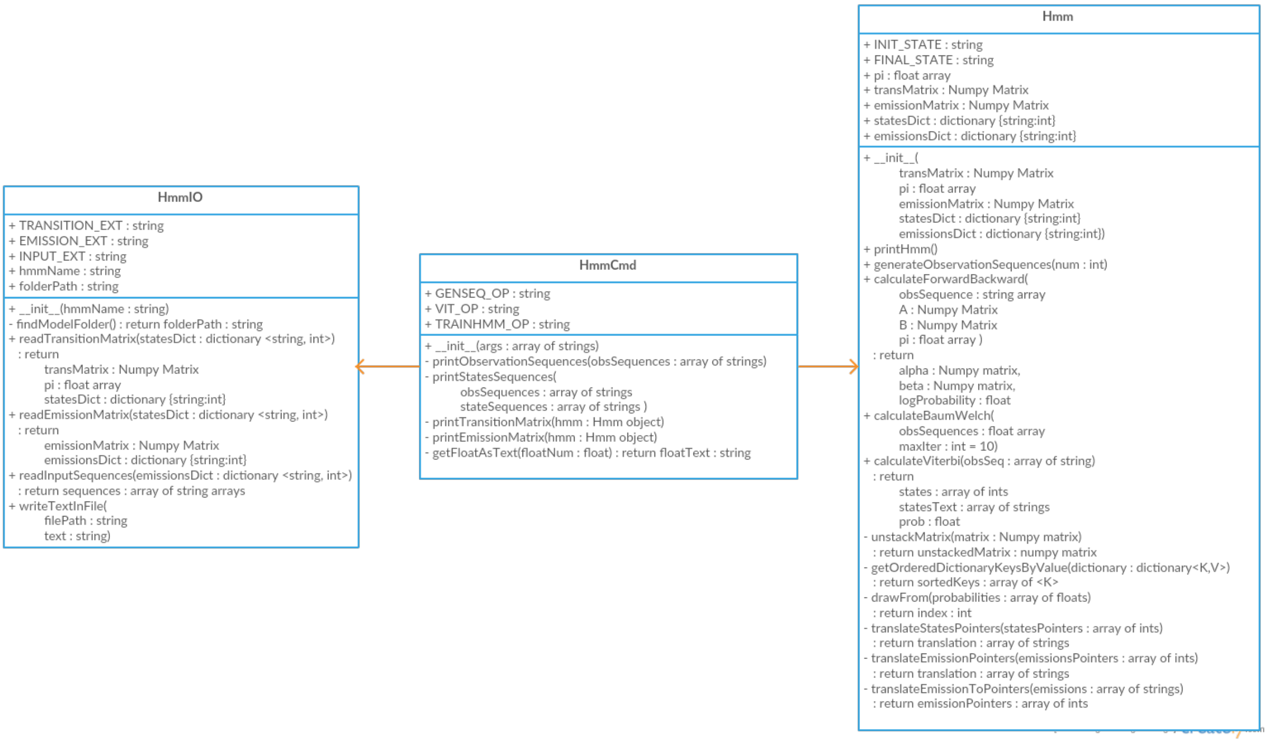

Figure 2: UML Class Diagram of the Solution

The available options for the command line program, along with the indications for using it, are described below, as specified by its help command of the program.

```
genseq:     Generates  a collection of observation sequences with each sequence on a line.
            It takes two parameters:
            - <name>    : the name of the HMM to work with
            - <num_seq> : (optional) the number of observation sequences to generate.
            Default: 10
            e.g. python HmmCmd.py genseq phone 12
            This program uses the files in the folder <name>:
            - <name>.trans : transition matrix structure
            - <name>.emit  : emission matrix structure
            As a result <num_seq> number of random observation sequences are shown based on the
            HMM structure.
vit:        The vit operation finds the most probable sequences of states based on given
            observation sequences, as well as their probability, using the Viterbi algorithm.
            It takes one parameter:
            - <name>    : the name of the HMM to work with
            e.g python HmmCmd.py vit phone
            This program uses the files in the folder <name>:
            - <name>.trans : transition matrix structure
            - <name>.emit  : emission matrix structure
            - <name>.input : file that contains the observation sequences
            As a result the most probable sequence of states with its probability is shown
```

```
                for each sequence.
trainhmm:       The trainhmm program trains the parameters of an HMM with a sequences of
                observations, using the Baum Welch algorithm.
                It takes two parameters:
                - <name>     : the name of the HMM to work with
                - <num_iter>: (optional) the maximum number iterations to run during
                training. Default: 10
                e.g. python HmmCmd.py trainhmm 10
                This program uses the files in the folder <name>:
                - <name>.trans : transition matrix structure with the apriori probabilities
                - <name>.emit  : emission matrix structure with the apriori probabilities
                - <name>.input : file that contains the observation sequences to train the HMM
                As a result the following files are created/overwritten, containing the structure
                of the trained HMM:
                - <name>_result.trans : the trained transition matrix structure
                - <name>_result.emit  : the trained emission matrix structure
--help:         Prints this help
```

4 Tests and Results

In order to test the results of solution implemented, two different models were designed. The first model **health** is based on a common problem for Hidden Markov Models, where in a village, whose villagers are either healthy or have a fever, a doctor diagnoses fever by asking patients how they feel (either normal, dizzy, or cold) [11]. The second problem is based on the Lin Hmm Implementation for C++ [5], where the model implements a phone call audio, with three different states (Onset, Mid, End) and for each state three possible emissions. A graphic visualization of these two models can be appreciated in Figures 3 and 4 respectively.

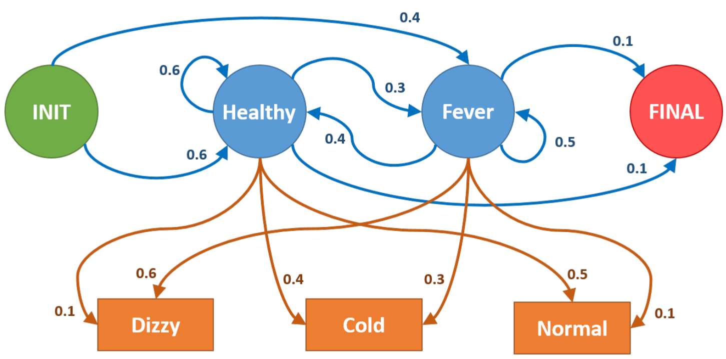

Figure 3: HMM Health Model

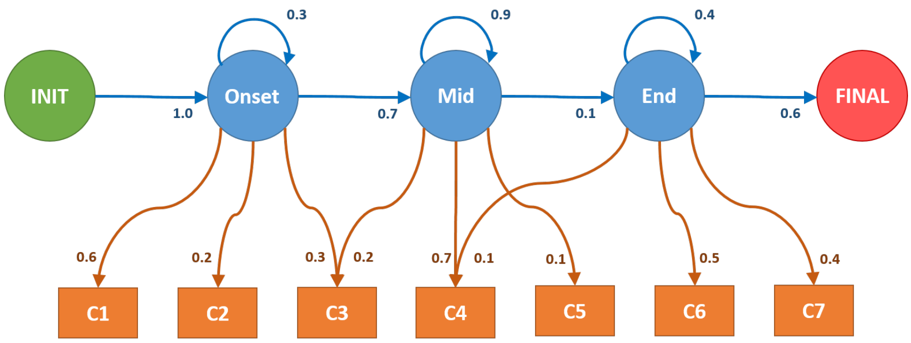

Figure 4: HMM Phone Model

Having these models defined, the three functionalities of the program were run and the outputs compared with the C++ Implementation of Hidden Markov Model [5], as is the aim of this program, explained in Section 1. The results of these functionalities are shown below, for each one of the three operations.

4.1 Generate Random Sequences with HMM Model

For this operation, and considering that its nature is random, the results are satisfactory and the implemented solution reproduce the original program correctly, producing random sequences of observations according to the proposed models. Figure 5 shows the output of both programs, when producing 5 random sequences, for the phone (above) and health (below) models.

phone model

```
>..\src\genseq.exe phone 10
C1 C4 C4 C4 C4 C4 C4 C4 C3 C4 C6 C7
C1 C3 C4 C4 C3 C4 C7
C2 C4 C4 C3 C4 C3 C3 C4 C4 C4 C4 C3 C4 C4 C7
C3 C1 C4 C5 C4 C4 C4 C4 C6
C1 C4 C4 C4 C4 C3 C3 C4 C4 C4 C4 C5 C3 C3 C5 C3 C3 C3 C4 C4 C3 C4 C4 C3 C4 C4 C4 C4 C4 C6
```

```
>python HmmCmd.py genseq phone 5
Using HMM in: ..\phone

C1 C3 C5 C3 C7 C4 C6
C2 C1 C4 C4 C7 C7 C6
C1 C5 C3 C4 C4 C4 C4 C4 C5 C3 C4 C4 C4 C3 C5 C4 C4 C4 C4 C3 C4 C4 C5 C3 C4 C7 C6 C4
C2 C3 C3 C4 C4 C4 C3 C5 C4 C3 C4 C4 C4 C5 C4 C4 C4 C3 C4 C7 C7
C3 C2 C1 C4 C4 C4 C4 C6
```

health model

```
>..\src\genseq.exe health 5
Normal Normal Cold Normal Cold Normal Dizzy Dizzy Cold Cold
Cold Dizzy Normal Normal Dizzy Dizzy Dizzy Cold Dizzy Normal Normal
Dizzy Cold Cold Dizzy Cold
Normal Cold Cold Normal Cold Normal Dizzy
Cold Normal Cold Dizzy Dizzy Normal Dizzy Dizzy Cold Dizzy Cold Normal Normal Dizzy Normal Dizzy
```

```
>python HmmCmd.py genseq health 5
Using HMM in: ..\health

Dizzy Normal Dizzy Cold Normal Dizzy Dizzy Cold
Cold Cold Normal Normal Normal Dizzy Normal Dizzy Cold Cold Cold Dizzy Normal
Dizzy Normal Cold Dizzy Dizzy Cold Cold
Dizzy Cold Dizzy Dizzy Dizzy Dizzy Normal
Normal Cold Dizzy Cold Normal Dizzy Normal
```

Figure 5: Results of the `genseq` operation for the phone and health models

4.2 Find the Most Probable Sequence of States with Viterbi

In order to test the implementation of the Viterbi algorithm, three sequences of observations were defined for each model, from which the most probable sequence of states is calculated. The sequences for each model can be seen below.

```
phone:    C1 C2 C3 C4 C4 C6 C7            health:   Normal Cold Dizzy
          C2 C2 C5 C4 C4 C6 C6                      Normal Cold Normal Cold Dizzy Dizzy
          C1 C2 C3 C4 C4 C6                         Cold Dizzy Dizzy Normal Cold Dizzy
```

phone model

```
>..\src\vit.exe phone


P(path)=0.625286
path:
C1  Onset
C2  Onset
C3  Mid
C4  Mid
C4  Mid
C6  End
C7  End
P(path)=0.936748
path:
C2  Onset
C2  Onset
C5  Mid
C4  Mid
C4  Mid
C6  End
C6  End
P(path)=0.625286
path:
C1  Onset
C2  Onset
C3  Mid
C4  Mid
C4  Mid
C6  End
```

```
>python HmmCmd.py vit phone
Using HMM in: ..\phone

P(path)=0.58982
path:
C1      Onset
C2      Onset
C3      Mid
C4      Mid
C4      Mid
C6      End
C7      End
P(path)=0.96899
path:
C2      Onset
C2      Onset
C5      Mid
C4      Mid
C4      Mid
C6      End
C6      End
P(path)=0.58982
path:
C1      Onset
C2      Onset
C3      Mid
C4      Mid
C4      Mid
C6      End
```

health model

```
>..\src\vit.exe health


P(path)=0.431482
path:
Normal  Healthy
Cold    Healthy
Dizzy   Fever
P(path)=0.290262
path:
Normal  Healthy
Cold    Healthy
Normal  Healthy
Cold    Healthy
Dizzy   Fever
Dizzy   Fever
P(path)=0.153887
path:
Cold    Healthy
Dizzy   Fever
Dizzy   Fever
Normal  Healthy
Cold    Healthy
Dizzy   Fever
```

```
>python HmmCmd.py vit health
Using HMM in: ..\health

P(path)=0.54545
path:
Normal  Healthy
Cold    Healthy
Dizzy   Fever
P(path)=0.31820
path:
Normal  Healthy
Cold    Healthy
Normal  Healthy
Cold    Healthy
Dizzy   Fever
Dizzy   Fever
P(path)=0.28877
path:
Cold    Healthy
Dizzy   Fever
Dizzy   Fever
Normal  Healthy
Cold    Healthy
Dizzy   Fever
```

Figure 6: Results of the `vit` operation for the phone and health models

The results of the implementation of the Viterbi algorithm, as explained in Subsection 2.1, can be appreciated in Figure 6. For this functionality, the solution implemented finds exactly the same most probable sequence of states in both models, although the calculated probabilities differ, in a small amount, form the original solution.

4.3 HMM Training with Baum-Welch

For training an HMM model using the Baum-Welch algorithm, along with the forward-backward algorithm, explained in Subsections 2.3 and 2.2 respectively, the initial probabilities of the models had to be defined a priory. This initial probabilities can be seen in Figure 7, where both the transition and emission probabilities are set to be equally distributed at the beginning.

phone init files

```
######### phone.trans #########
INIT
INIT      Onset     1
Onset     Onset     0.5
Onset     Mid       0.5
Mid       Mid       0.5
Mid       End       0.5
End       End       0.5
End       FINAL     0.5
```

```
######### phone.emit #########
Onset     C1        0.33
Onset     C2        0.33
Onset     C3        0.33
Mid       C3        0.33
Mid       C4        0.33
Mid       C5        0.33
End       C4        0.33
End       C6        0.33
End       C7        0.33
```

health init files

```
######### health.trans #########
INIT
INIT      Healthy   0.5
INIT      Fever     0.5
Healthy   Healthy   0.33
Healthy   Fever     0.33
Healthy   FINAL     0.33
Fever     Healthy   0.33
Fever     Fever     0.33
Fever     FINAL     0.33
```

```
######### health.emit #########
Healthy   Dizzy     0.33
Healthy   Cold      0.33
Healthy   Normal    0.33
Fever     Dizzy     0.33
Fever     Cold      0.33
Fever     Normal    0.33
```

Figure 7: Configuration files for Initial Probabilities of the Models

With the initial parameters defined, the Baum-Welch algorithm is run and its results can be appreciated in Figure 8. It is important to notice that the results of both solutions are comparable and the probabilities only vary in a small amount.

phone model

```
###### Console Output ######
>..\src\trainhmm.exe phone
phone_bw_result phone_bw.input

Training with Baum-Welch for up to
10 iterations, using 100 sequences.
Iteration 1 totalLogProb=-1505.32
Iteration 2 totalLogProb=-1501.94
Iteration 3 totalLogProb=-1502.3

###### phone_bw_result.trans ######
INIT
INIT Onset 1
Onset Onset 0.367824
Onset Mid 0.632176
Mid Mid 0.895835
Mid End 0.0979835
End End 0.429223

######  phone_bw_result.emit ######
Onset C1 0.448845
Onset C2 0.21494
Onset C3 0.336215
Mid C3 0.197722
Mid C4 0.711654
Mid C5 0.0906237
End C4 0.0655485
End C6 0.533972
End C7 0.400479
```

```
###### Console Output ######
>HmmCmd.py trainhmm phone_bw
Using HMM in: ..\phone_bw

Training with Baum-Welch for up to
10 iterations, using 100 sequences.
Iteration 1 totalLogProb=-2145.66
Iteration 2 totalLogProb=-1505.36
Iteration 3 totalLogProb=-1498.51

###### phone_bw_result.trans ######
INIT
INIT    Onset   1
Onset   Onset   0.371381
Onset   Mid     0.628619
Mid     Mid     0.895193
Mid     End     0.098688
End     End     0.445415
End     FINAL   0.554585

######  phone_bw_result.emit ######
Onset   C1      0.446319
Onset   C2      0.21373
Onset   C3      0.33995
Mid     C3      0.198003
Mid     C4      0.710815
Mid     C5      0.091182
End     C4      0.092983
End     C6      0.518295
End     C7      0.388722
```

health model

```
###### Console Output ######
>..\src\trainhmm.exe health_bw
health_bw-results health_bw.input

Training with Baum-Welch for up to
10 iterations, using 100 sequences.
Iteration 1 totalLogProb=-1369.84
Iteration 2 totalLogProb=-1111.17
Iteration 3 totalLogProb=-1111.17

###### health_bw_result.trans ######
INIT
INIT Healthy 0.5
INIT Fever 0.5
Healthy Healthy 0.446004
Healthy Fever 0.446004
Fever Healthy 0.446004
Fever Fever 0.446004

######  health_bw_result.emit ######
Healthy Dizzy 0.343413
Healthy Cold 0.339093
Healthy Normal 0.317495
Fever Dizzy 0.343413
Fever Cold 0.339093
Fever Normal 0.317495
```

```
###### Console Output ######
>HmmCmd.py trainhmm health_bw
Using HMM in: ..\health_bw

Training with Baum-Welch for up to
10 iterations, using 100 sequences.
Iteration 1 totalLogProb=-1369.84
Iteration 2 totalLogProb=-1111.17
Iteration 3 totalLogProb=-1111.17

###### health_bw_result.trans ######
INIT
INIT    Healthy 0.5
INIT    Fever   0.5
Healthy Healthy 0.446004
Healthy Fever   0.446004
Fever   Healthy 0.446004
Fever   Fever   0.446004
Fever   FINAL   0.107991

######  health_bw_result.emit ######
Healthy Dizzy   0.343413
Healthy Cold    0.339093
Healthy Normal  0.317495
Fever   Dizzy   0.343413
Fever   Cold    0.339093
Fever   Normal  0.317495
```

Figure 8: Results of the `trainhmm` operation for the phone and health models

5 Conclusions

The developed program presents a correct solution for the most common challenges for the Hidden Markov Model. The proposed design, described in Section 3, correctly distributes the responsibilities of the program, and it standardizes its usability, compared with the original solution by Lin, specially with the file management and the process to call the program's operations.

As shown in Section 4, the presented program is able to generate random observation sequences, present the most probable sequence of states given a sequence of observations and learn the parameters of a Hidden Markov Model given a set of observation sequences. This includes a python-based design of an HMM, along with the implementation of the Viterbi, forward-backward and Baum-Welch algorithms, and its results are satisfactory, being compared with the C++ Implementation of Hidden Markov Model, by Dekang Lin [5].

References

[1] Roger D. Boyle. Forward-Backward Algorithm. http://www.comp.leeds.ac.uk/roger/HiddenMarkovModels/html_dev/forward_backward/s1_pg1.html. [Online; accessed 30-01-2017].

[2] NumPy developers. Numpy. http://www.numpy.org/, 2016. [Online; accessed 30-01-2017].

[3] Python Software Foundation. Python 2.7.0 release. https://www.python.org/download/releases/2.7/, 2017. [Online; accessed 30-01-2017].

[4] Christian Kohlschein. An introduction to hidden markov models. http://www.tcs.rwth-aachen.de/lehre/PRICS/WS2006/kohlschein.pdf. [Online; accessed 30-01-2017].

[5] Dekang Lin. A C++ Implementation of Hidden Markov Model. http://stp.lingfil.uu.se/~starback/hmm/, 2011. [Online; accessed 30-01-2017].

[6] Luz Abril Torres Méndez. The Viterbi Algorithm. http://www.cim.mcgill.ca/~latorres/Viterbi/va_alg.htm. [Online; accessed 30-01-2017].

[7] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[8] Wikipedia. Baum–Welch algorithm. https://en.wikipedia.org/wiki/Baum-Welch_algorithm, 2017. [Online; accessed 30-01-2017].

[9] Wikipedia. Forward–backward algorithm. https://en.wikipedia.org/wiki/Forward-backward_algorithm, 2017. [Online; accessed 30-01-2017].

[10] Wikipedia. Hidden Markov model. https://en.wikipedia.org/wiki/Hidden_Markov_model, 2017. [Online; accessed 30-01-2017].

[11] Wikipedia. Viterbi algorithm. https://en.wikipedia.org/wiki/Viterbi_algorithm, 2017. [Online; accessed 30-01-2017].