Implementing Canonical Baum-Welch and Viterbi Algorithms for Hidden Markov Model to Generate Synthetic DNA Sequences

(Dora) Dongshunyi Li, (Eden) Huang Huang

Abstract:

Our project reviews a selected bioinformatics academic paper on Hidden Markov Model (HMM) with focus on the implementation of canonical Viterbi, forward-backward and Baum-Welch Algorithms in Python on simulated and real data. We further explore the options of codes optimization discussed in the capstone course STA 663 Statistical Computation, including the discussion of better algorithms and data structures, the JIT compilation of critical functions, vectorization and parallelism.

Then we apply the three target algorithms on two datasets. For both datasets, we generate synthetic DNA sequences with Hidden Markov Models. The datasets were obtained by our team member Dora Li who is conducting an ongoing research in bioinformatics using the same datasets. The data source is yet to be publicized.

The project write-up includes five sections: **Section 1 -- Background and Data** describes background of the selected research paper and the dataset for application, as well as a summary of the submitted codes and files on GitHub repository. **Section 2 -- Theory** recaps the mathematical concepts of Hidden Markov Model the three target algorithms with examples. **Section 3 -- Implementation and Optimization** documents the improvement in performance with optimization performed. **Section 4 -- Results and Testing** shows the results on genome prediction. We also conducted a comparative analysis (speed and accuracy) with competing algorihtms from Python 3 native libary `hmmlearn` on a simulated data. Testing is illustrated in this section as well **Section 5 -- Discussion** concludes with discussion.

Section 1 -- Background and Data

(i). Overview

A Hidden Markov Model (HMM) is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states. It provides a powerful application in speech recognition, bioinformatics, knowledge discovery and clustering.

The selected paper is *Implementing EM and Viterbi algorithms for Hidden Markov Model in Linear Memory* by Alexander Churbanov and Stephen Winters-Hilt from *BMC Bioinformatics*. This paper compares the computational expenses of canonical, checkpointing and linear implementations of Baum-Welch learning, Viterbi decoding and forward-Backword algorithms. It concludes that compared to the canonical and linear methods, the checkpointing implementations produce best overall tradeoff between memory use and speed. The linear modification replaces the array with linkedlist to store the dynamic programming backtrack pointers in viterbi algorithm and improve the efficiency of memory use. However, the original paper chooses java for implementation that has built-in linkedlist data structure. Furthermore, the linear and checkpointing methods would not boost the time performance. For simplicity purpose, our implementation in Python, which does not directly support linkedlist, focuses on the canonical method.

(ii). GitHub and Codes

repository link: https://github.com/edenhuangSH/STA663_Final_Project (https://github.com/edenhuangSH/STA663_Final_Project)

The codes of this project is available in a public GitHub repository with a README file, an open source license, source code, test code, examples and the reproducible report. The output package `linearhmm` is installable with `python setup.py install`. `linearhmm` includes two major methods:

- **utils.py**
  - Helper functions to generate random value given discrete distribution and encode DNA to integers
- **lineaerhmm.py**
  - Implement Viterbi Algorithm to maximize a posteri
  - Implement Forward-Backward Algorithm for posterior marginals
  - Implement Baum Welch Algorithm for expectation maximization inference

For other files, documents and notes, please see the repository.

(iii). Introduction to Implementation on DNA Data

The application of the three algorithms on the real data is to generate DNA mutations following a particular mutation spectra for whole-genome sequencing studies. We first train our Hidden Markov Models on sequencing data using Viterbi algorithm from cancer patients, then evaluate likelihood using forward-backword algorithm, and finally predict the models on the reference genome. The final output are the transition and emisstion matrices learned via the Baum-Welch algorithm as well as the decoding for a most possible path found by the Viterbi algorithm.

Section 2 -- Theory

(i). Concepts of Hidden Markov Models

Intuition:

- Undirected graphical model.
- Connections between nodes indicate dependence.
- We observe Y_1 through Y_n, which we model as being observed from hidden states S_1 through S_n.
- Any particular state variable S_k depends only on S_{k-1} (what came before it), S_{k+1} (what comes after it), and Y_k (the observation associated with it).

Model Parameter:

- **Transition distribution:**
  - describes the distribution for the next state given the current state.
  - $P(nextState|currentState)$

- **Emission distribution:**
  - describes the distribution for the output given the current state.
  - $P(Observation|currentState)$

- **Initial state distribution:**
  - describes the starting distribution over states.
  - $P(initialState)$

(ii). Concepts of Viterbi Algorithm

The Viterbi algorithm finds the most likely series of states, given some observations and assumed parameters.

Intuition:

- Efficient way of finding the most likely state sequence.
- Method is general statistical framework of compound decision theory.
- Maximizes a posteriori probability recursively.
- Assumed to have a finite-state discrete-time Markov process.

Pseudocodes and Equations:

- Maximum a posteri (MAP) probability, given by:
$$P(States|Observations) = \frac{P(Observations)|States)P(States)}{P(Observations)}$$
- Given a hidden Markov model (HMM) with:
  - State space S
  - Initial probabilities π_i of being in state i
  - Transition probabilities $a_{i,j}$ of transitioning from state i to state j.
  - We observe outputs $y_1, \ldots, y_T$.
  - The most likely state sequence $x_1, \ldots, x_T$ that produces the observations is given by the recurrence relations:
$$V_{1,k} = P(y_1|k) \cdot \pi_k$$
$$V_{t,k} = max_x(P(y_t|k) \cdot a_{x,k} \cdot V_{t-1,x}$$
- $V_{t,k}$ is the probability of the most probable state sequence $\mathrm{P}\big(x_1, \ldots, x_T, y_1, \ldots, y_T\big)$
- The Viterbi path can be retrieved by saving back pointers that remember which state x was used in the second equation.
- Let $\mathrm{Ptr}(k, t)$ be the function that returns the value of x used to compute $V_{t,k}$ if $t > 1$, or k if $t = 1$. Then:

$$x_T = argmax_x(V_{T,x})$$

$$x_{t-1} = Ptr(x_t, t)$$

(iii). Concepts of Forward-backward Algorithm

The goal of the forward-backward algorithm is to find the conditional distribution over hidden states given the data.

Intuition:

- It is used to find the most likely state for any point in time.
- It cannot, however, be used to find the most likely sequence of states

Pseudocodes and Intuition:

- Computes posterior marginals of all hidden state variables given a sequence of observations/emissions.
- Computes, for all hidden state variables $S_k \in \{S_1, \dots, S_t\}$, the distribution $P(S_k \mid o_{1:t})$. This inference task is usually called smoothing.
- The algorithm makes use of the principle of dynamic programming to compute efficiently the values that are required to obtain the posterior marginal distributions in two passes.
- The first pass goes forward in time while the second goes backward in time.

(iv). Concepts of Baum Welch Algorithm:

Baum–Welch algorithm is used to infer unknown parameters of a Hidden Markov Model. It makes use of the forward-backward algorithm to update the hypothesis.

Model Parameters:

- Initial State Probabilities
- Transition Matrix
- Emission Matrix

Pseudocodes and Equations:

- Calculate the temporary variables, according to Bayes' theorem:
  Temporary variables are the probabilities of being in state i at time t given the observed sequence Y and the parameters θ
  $$\gamma_i(t) = P(X_t = i|Y, \theta) = \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N} \alpha_j(t)\beta_j(t)}$$
  The probability of being in state i and j at times t and $t+1$ respectively given the observed sequence Y and parameters θ:
  $$\xi_{ij}(t) = P(X_t = i, X_{t+1} = j|Y, \theta) = \frac{\alpha_i(t)a_{ij}\beta_j(t+1)b_j(y_{t+1})}{\sum_{k=1}^{N} \alpha_k(T)}$$
- Update Inital State Probability:
  Initial state probabilities are the expected frequency spent in state i at time 1:
  $$\pi_i^* = \gamma_i(1)$$
- Update Transition Matrix:
  Transition matrix describes the expected number of transitions from state i to state j compared to the expected total number of transitions away from state i:
  $$a_{ij}^* = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)}$$

- Update Emission Matrix:
  Emission matrix documents the expected number of times the output observations have been equal to v_k while in state i over the expected total number of times in state i:
  $$b_i^*(v_k) = \frac{\sum_{t=1}^{T} 1_{y_t=v_k} \gamma_i(t)}{\sum_{t=1}^{T} \gamma_i(t)}$$

Section 3 -- Implementation and Optimization

The codes for this section is uploaded at: **repository link**:
https://github.com/edenhuangSH/STA663_Final_Project
(https://github.com/edenhuangSH/STA663_Final_Project)

The main functionality of our package is learning a Hidden Markov Model given a sequence of observations and decoding a given sequence of observations to get the most likely sequence of states, given a Hidden Markov Model. The first function is implemented through Baum-welch algorithm and the second is through Viterbi algorithm. We first implemented the algorithms using pain Python with Numpy arrays and matrixes. We profile the code and found the part consuming most of time is the function implementing Baum-welch algorithm to fit a HMM on a sequence of observations. According to the profiling results, the part consumes time most is a helper function called by this function. The helper function is used to estimate the probability for a given sequence of observations and a HMM. This function is called every time the E-M algorithm (Baum-welch algorithm) updates its parameters till converges. Given that it is called multiple times and there is a lot matrix manipulations in the helper function, we decided to mainly optimize this helper function.

We start by profiling this helper function. It calls on average 29ms to run. The part consumes time most is the function "**array**finalize" and "**new**" in defmatrix.py of the matrixlib of Numpy. These 2 functions called whenever a matrix is initialized. We implemented a lot np.matrix() to make our 2D arrays matrix so as to apply matrix manipulations. However, we soon find out that this is not necessary, given that many matrix manipulations can be done on 2D-array. Therefore, we remove all unnecessary np.matrix(). After this is done, the helper function was sped up to 28ms.

In our implementation, the data structures used are Numpy arrays and lists. These 2 are considered the best and optimal data structures for the functions they serve. So there is not much to optimize with the data structures. Then we proceed to Cython. As illustrated before, there is a lot of matrix manipulations and algorithmic operations in our helper function. These can be done much faster in C instead of Python. We use Cython magic in Jupyter to profile the code. We declared the variables and replaced all the Numpy matrix manipulations with our our own matrix manipulation functions written in Cython. This includes matrix multiplications, element-wise multiplications/divisions and summation along axis in a 2D matrix. We applied these user-defined functions to our helper function, resulting in a running time of 20s.

Section 4 -- Results and Testing

(i). Summary of Results

-- Results on a Simulated Dataset:

In [101]:
```
import random
import numpy as np
import linearhmm
states = ('AG_rich', 'CT_rich')
observations = (0, 1, 2, 3)
start_probability = {'AG_rich': 0.6, 'CT_rich': 0.4}

transition_probability = {
   'AG_rich' : {'AG_rich': 0.8, 'CT_rich': 0.2},
   'CT_rich' : {'AG_rich': 0.4, 'CT_rich': 0.6}
   }

emission_probability = {
   'AG_rich' : {2: 0.5, 1: 0.1, 3: 0.3, 0:0.1},
   'CT_rich' : {2: 0.1, 1: 0.4, 3: 0.1, 0:0.4}
   }
N = 100
hidden, visible = linearhmm.simulate_data(N,states, observations, start_prob
```

Get the accuracy from Viterbi

In [102]:
```
(prob, p_hidden) = linearhmm.viterbi(visible,
                   states,
                   start_probability,
                   transition_probability,
                   emission_probability)
# assess accuracy of the HMM model
wrong= 0
for i in range(len(hidden)):
    if hidden[i] != p_hidden[i]:
        wrong = wrong + 1
print ("accuracy: " + str(1-float(wrong)/N))
```

```
accuracy: 0.91
```

Get the transition and emission matrixes learned by Baum-Welch algorithm

In [107]:
```
A_mat, O_mat = linearhmm.baum_welch(num_states=2,num_obs=4,observ=np.array(v
print(A_mat)
print(O_mat)
```

```
[[ 0.5  0.5]
 [ 0.5  0.5]]
[[ 0.15142857  0.29142857  0.1         0.45714286]
 [ 0.15142857  0.29142857  0.1         0.45714286]]
```

-- Results on the sample of our DNA Dataset:

DNA is composed of different states of nucleotides, i.e. A, C, T, G. Publications have shown that contiguous nucleotides are highly dependent with each other. In other words, a nucleotide determines if the nucleotide after it is A, C, T or G. Therefore, the generation of synthetic DNA can be modeled as a Markov chain, where there are 4 possible observations. However, it was further revealed that the emission probabilities in different regions of DNA is different. For example, the probability for a A after a C is different in A-G rich region and in C-T rich region. In this case, a Hidden Markov Model is more appropriate where there are 2 states $S = S_1, S_2$, with S_1 and S_2 representing A-G rich and C-T rich. Each state has 4 possible observations and the emission probabilities can be modeled as $b_j(o_t) = p(o_t|q_t = S_j)$. Here $1 \leq j \leq N$, where N is the total number of states and equal to 2 in our case. t is a time point and we have $1 \leq t \leq T$, where T is the total number of time points. q_t is a realized state being visited at time t.

The operations mainly involve manipulations of DNA strings. Operations on strings (e.g. comparison, concatenation) are more costly in terms of both time and space, compared to algorithmtic operations. Therefore, one approach is to encode DNA sequences as binaries or integers. A single nucleotide can be encoded as a 2-digit binary string or a single integer. In this way, all string manipulations can be converted to algorithmic operations. The relationship between nucleotide and binary/decimal representation is shown below:

Nucleotides	Binary String	Integer
T	00	0
C	01	1
A	10	2
G	11	3

Following the modeling methods described above, we applied our implementations of the Baum-Welch algorithm and the Viterbi algorithm on a simulated data set and on a real data set. We will show the transition and emisstion matrixes learned via the Baum-Welch algorithm and also the decoding for a most possible path found by the Viterbi algorithm.

The actual implementation is on the sample of real data set, i.e. sequence of 350 length of the reference genome. It is extracted from the promoter region:

In [106]:
```
raw = 'GCTGCGGGGAGGGGGGCGCGGGTCCGCAGTGGGGATGTGCTGCCGGGAGGGGGGCGCGGGTCCGCAGTG
visible = list(raw)

def encode_DNA(x):
    """convert base to integer representation"""
    if x == 'T':
        num = 0
    elif x == 'C':
        num = 1
    elif x == 'A':
        num = 2
    elif x == 'G':
        num = 3
    else:
        num = 4

    return num
visible = [encode_DNA(x) for x in visible]
```

Get the transition and emission matrixes learned by Baum-Welch algorithm

In [109]:
```
A_mat, O_mat = linearhmm.baum_welch(num_states=2,num_obs=4,observ=np.array(v
print(A_mat)
print(O_mat)
```

```
[[ 0.5  0.5]
 [ 0.5  0.5]]
[[ 0.15142857  0.29142857  0.1         0.45714286]
 [ 0.15142857  0.29142857  0.1         0.45714286]]
```

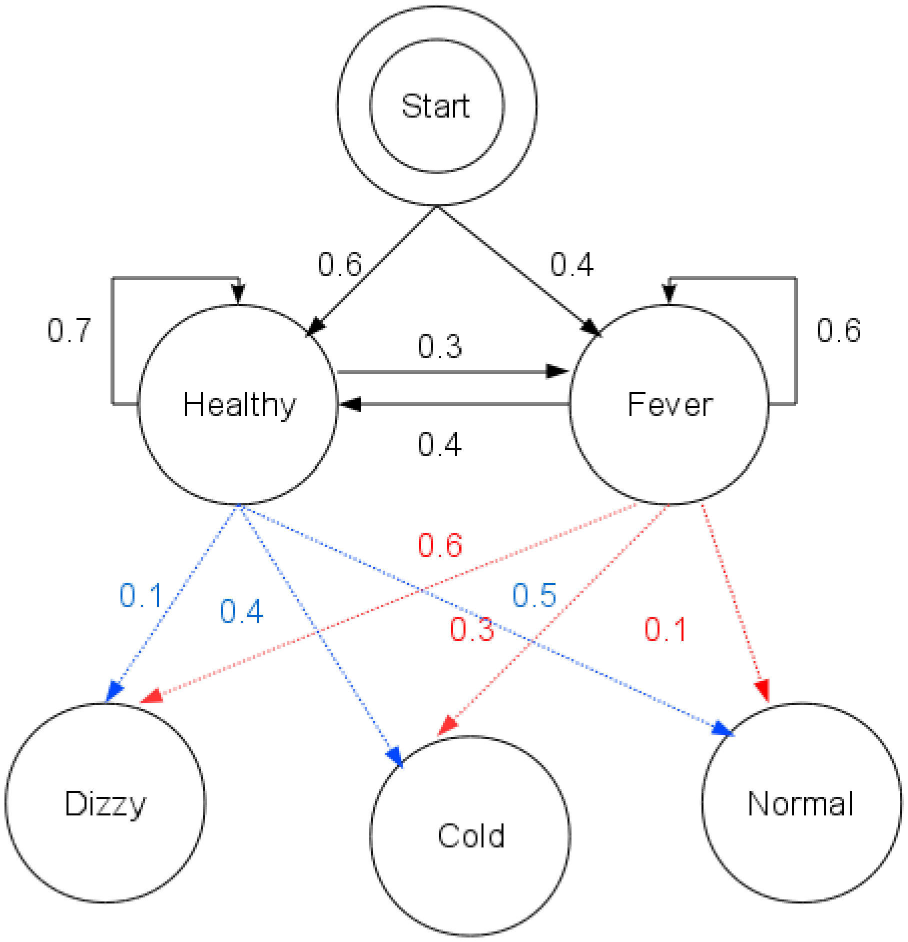
```
The above results show that our viterbi algorithm is giving good accuracy
(about 80%) in predicting genome sequence
```

(ii). Comparative Analysis with `hmmlearn` using an example

Let's consider the following simple HMM.

- Composed of 2 hidden states: Healthy and Fever.
- Composed of 3 possible observation: Normal, Cold, Dizzy

The model can then be used to predict if a person is feverish at every timestep from a given observation sequence. There are several paths through the hidden states (Healthy and Fever) that lead to the given sequence, but they do not have the same probability.



(**source**: Havard AM207 Course Website)

-- Implementation on `linearhmm` (our algorithms):

a) Generate simulated data set

In [88]:
```
import random
import numpy as np
import linearhmm
np.random.seed(1)

states = ('Healthy', 'Fever')
observations = ('cold', 'normal', 'dizzy')
start_probability = {'Healthy': 0.6, 'Fever': 0.4}

transition_probability = {
   'Healthy' : {'Healthy': 0.69, 'Fever': 0.3, 'E': 0.01},
   'Fever' : {'Healthy': 0.4, 'Fever': 0.59, 'E': 0.01},
   }

emission_probability = {
   'Healthy' : {'normal': 0.5, 'cold': 0.4, 'dizzy': 0.1},
   'Fever' : {'normal': 0.1, 'cold': 0.3, 'dizzy': 0.6},
   }

N = 100 # 100 samples
hidden, visible = linearhmm.simulate_data(N, states, observations, start_pro
```

b) Get the accuracy from Viterbi Algorithm:

In [89]:
```
(prob, p_hidden) = linearhmm.viterbi(visible,
                   states,
                   start_probability,
                   transition_probability,
                   emission_probability)

# assess accuracy of the HMM model:
wrong= 0
for i in range(len(hidden)):
    if hidden[i] != p_hidden[i]:
        wrong = wrong + 1
print ("accuracy: " + str(1-float(wrong)/N))
```

```
accuracy: 0.78
```

-- Implementation on `hmmlearn` (Python 3 library):

In [83]:
```
import random
import numpy as np
import hmmlearn.hmm as hmm
np.random.seed(1)

# same model parameters as previous:
model = hmm.MultinomialHMM(n_components=2) # 2 states
# Initial population probability:
model.startprob_ = np.array([0.6, 0.4])
# The transition matrix between component 1 and 2:
model.transmat_ = np.array([[0.7, 0.3],
                            [0.4, 0.6]])
model.emissionprob_ = np.array([[0.5, 0.4, 0.1],
                                [0.1, 0.3, 0.6]])

# use the same prediction set from previous simulated data (converting state
def encode_states(x):
    """convert states to integer representation"""
    if x == 'cold':
        num = 0
    elif x == 'normal':
        num = 1
    elif x == 'dizzy':
        num = 2
    return num

visible_obs = [encode_states(x) for x in visible]
visible_obs = np.array([visible_obs]).T
model = model.fit(visible_obs)
A_mat, O_mat = model.decode(visible_obs, algorithm="viterbi")

# assess accuracy of the HMM model under hmmlearn implementation:
wrong= 0
for i in range(len(hidden)):
    if hidden[i] != p_hidden[i]:
        wrong = wrong + 1
print ("accuracy: " + str(1-float(wrong)/N))
```

```
accuracy: 0.78
```

We can see that compared to the viterbi algorithms from our package `linearhmm`, `hmmlearn` library implementation would give the same results and accuracy rate.

(iii). Codes Testing

We tested whether or not our functions would generate desirable output using the test codes. The test classes were given by wikipedia (please see reference for more details). We also uploaded the test codes folder on GitHub. The following examples test random number generator and viterbi functions. The results show that we successfully pass these tests and the codes are working correctly.

In [97]:
```
%%writefile test_utils.py

import numpy as np
from utils import get_discrete_value


def test_get_discrete_value():

    random.seed(1234)
    values = (0, 1, 2, 3)
    probabilities= np.array([0.25, 0.25, 0.25, 0.25])

    assert get_discrete_value(values, probabilities) == 3
```

```
Overwriting test_utils.py
```

In [98]:
```
!python test_utils.py
```

In [95]:
```
%%writefile test_HMMchain.py

from HMMchain import HMMchain

def setUp(self):
    self.states = ('AG_rich', 'CT_rich')
    self.observations = (0, 1, 2, 3)
    self.start_probability = {'AG_rich': 0.6, 'CT_rich': 0.4}
    self.transition_probability = {
                                   'AG_rich' : {'AG_rich': 0.8, 'CT_rich': (
                                   'CT_rich' : {'AG_rich': 0.4, 'CT_rich': (
                                  }
    self.emission_probability = {
                                   'AG_rich' : {2: 0.5, 1: 0.1, 3: 0.3, 0:0.
                                   'CT_rich' : {2: 0.1, 1: 0.4, 3: 0.1, 0:0.
                                }
    self.N = 10


def test_simulate(self):
    res0, res1 = (['CT_rich',
  'CT_rich',
  'CT_rich',
  'AG_rich',
  'AG_rich',
  'CT_rich',
  'CT_rich',
  'AG_rich',
  'AG_rich',
  'AG_rich'],
 [3, 1, 0, 2, 2, 1, 0, 2, 0, 3])
    self.assertEqual((res0, res1))
```

```
Overwriting test_HMMchain.py
```

In [96]:
```
!python test_HMMchain.py
```

Section 5 -- Discussion

According to the **section 3** and **section 4**, we successfully reproduce powerful viterbi, backward-forward and Baum-Welch algorithms with desirable performance and accuracy in predicting genome sequence. Due to the time limit of the project, we did not scale our codes for massive data sets or write C/C++ functions to further polish optimization and other applications. These material would be carried on for future studies in other courses or internship experience.

Our two team members, Dora Li and Eden Huang split all the tasks evenly throughout the project. Both team members worked together on algorithms coding, literature review and junit testing. Dora was also responsible for codes optimization, parallel programming and real data application, while Eden focused on concepts overview, comparative analysis project write-up and results reporting.

Reference:

1.Churbanov, A., & Winters-Hilt, S. (2008). "Implementing EM and Viterbi algorithms for Hidden Markov Model in linear memory". BMC Bioinformatics, 9(224). Retrieved April 20, 2017.

2.Yuan, J.J. (2014, Jan 22). "VITERBI ALGORITHM: FINDING MOST LIKELY SEQUENCE IN HMM" [Web log post]. Retrieved from https://jyyuan.wordpress.com/2014/01/28/baum-welch-algorithm-finding-parameters-for-our-hmm/ (https://jyyuan.wordpress.com/2014/01/28/baum-welch-algorithm-finding-parameters-for-our-hmm/)

3.Yuan, J.J. (2014, Jan 26). "FORWARD-BACKWARD ALGORITHM: FINDING PROBABILITY OF STATES AT EACH TIME STEP IN AN HMM" [Web log post]. Retrieved from https://jyyuan.wordpress.com/2014/01/26/forward-backward-algorithm-finding-probability-of-states-at-each-time-step-in-an-hmm/ (https://jyyuan.wordpress.com/2014/01/26/forward-backward-algorithm-finding-probability-of-states-at-each-time-step-in-an-hmm/)

4.Yuan, J.J. (2014, Jan 28). "BAUM-WELCH ALGORITHM: FINDING PARAMETERS FOR OUR HMM" [Web log post]. Retrieved from https://jyyuan.wordpress.com/2014/01/28/baum-welch-algorithm-finding-parameters-for-our-hmm/ (https://jyyuan.wordpress.com/2014/01/28/baum-welch-algorithm-finding-parameters-for-our-hmm/)

5.Baum, L. E.; Petrie, T. (1966). "Statistical Inference for Probabilistic Functions of Finite State Markov Chains". The Annals of Mathematical Statistics. 37 (6): 1554–1563. doi:10.1214/aoms/1177699147. Retrieved 28 April 2017.

6.Rabiner, Lawrence. "First Hand: The Hidden Markov Model". IEEE Global History Network. Retrieved 28 April 2017.

7.Protopapas, Pavlos. Lecture 18 and Lecture 19, AM207 Spring 2014. Havard Extension course website. Retrieved from http://iacs-courses.seas.harvard.edu/courses/am207/blog/lecture-18.html (http://iacs-courses.seas.harvard.edu/courses/am207/blog/lecture-18.html)