SOFTWARE

Hidden Markov Model : Prediction of protein secondary structure

Hyejeong Cheon

Correspondence:
hyejeonc@stud.ntnu.no
Department of physics, NTNU, Trondheim , Norway
Full list of author information is available at the end of the article

Abstract

Background: The prediction of protein secondary structure is one of the most popular study fields in bioinformatics and computational biology. The prediction is important because understanding secondary structure of protein is related to how it behaves as chemical/biological reactor in our body. Prediction of protein secondary structure is not easy and determined from properties of each amino acids and environment. Here, one of machine learning methods with Hidden Markov Model is used and the model is trained by expectation maximization(EM) algorithm and simple counting method for predicting protein secondary structure.

Conclusion: HMM is useful to predict protein secondary structure. HMM with EM algorithms is more beneficial to predict protein secondary structure than the HMM combining the simple counting method. Especially, the EM HMM with maximum iteration shows high accuracy for three possible secondary structure.

Keywords: Hidden Markov Model; Protein secondary structure prediction; secondary structure

Introduction

Protein secondary structure prediction

A protein is composed of multiple amino acids and the sequence of amino acids is important in a perspective of properties and functions of a protein. The interactions, for example, hydrogen bond, Van der Waals interaction, electrostatic interaction of amino acids can affect the secondary structure of protein. That means, not only secondary structure but also tertitary structure and the final 3D conformation of protein can be influenced. There are existent algorithms already for predicting protein secondary structures, for examples, one of these method is known as PSI-Blast(Position-Specific Iterated BLAST), using BLAST(Basic Local Alignment Search Tool) algorithm which refers a method of comparing a query sequence with protein sequences in database with a scoring matrix(substitution matrix). [1]

Machine learning algorithm

There are three types of machine learning algorithms. The first is supervised learning, second is unsupervised learning and third is reinforcement learning. They are differentiated with whether training data includes desired output or not and whether training state is dynamic or static. Supervised learning is used here for predicting protein secondary structures. Train sets include protein sequences and secondary structures, known as 'training data'.

Background

Hidden Markov Model

Markov model is a mathematical model that refers present state is only dependent on previous state with transition probabilities, regardless of other states before the previous state. This can be explained as below,

$$P(q_i|q_1, q_2, ..., q_{i-1}) = p(q_i|q_{i-1})$$

where q is a possible state at ith state(step). Hidden Markov model(HMM) is a Markov model with the same concept that the present state is only dependent on the previous state, but states are hidden and users confirm differences between states by observations, and the observations are determined by emission probabilities with each present state. This is described in figure 2.

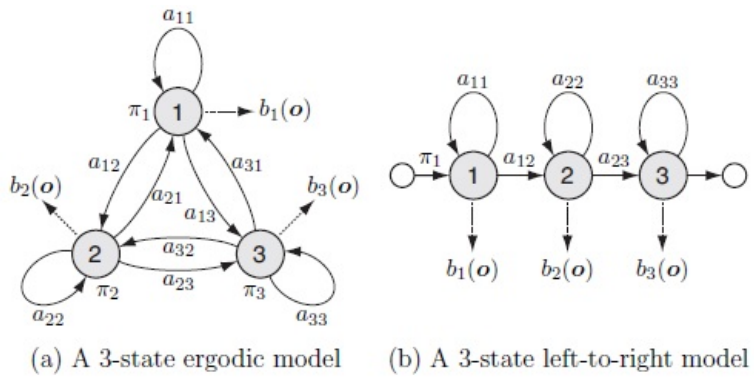

Figure 1 Concept of Hiddem Markov Model(HMM) This figure depicts what HMM process is with o, observation(or symbol), a, transition probability, b, emission probability and π, start probability with $1, 2, 3$ state. (a) 3-state ergodic model is implemented here. [2]

Therefore, HMM is explained with a set of three parameters, (π, A, B), π is an initial distribution of the first states, A denotes transition probability from states to states and B denotes emission probability from states to observations(symbols).

Here, we describe the available secondary structures of a protein with a amino acid sequence as three types, 'h' as α-helix, 'e' as β-sheet and 'c' as random-coil, as the hidden states following Markov chain. The possible observations are 20 amino acids symbols of protein. 20 amino acids(symbols) are simplified as 20 alphabet letter symbols as described in table 1. The main idea is taking a linear sequence of amino acids and predicting which secondary structure state corresponds to each amino acid.

Algorithm

System and model

We define a HMM with a specific set, (π, A, B), π is an start probability of three 'h', 'e', 'c' states, A denotes transition probabilities from three 'h', 'e', 'c' states to three 'h', 'e', 'c' states and B denotes emission probabilities from three 'h', 'e', 'c' states to twenty observations(symbols).

Table 1 Table of 20 amino acids shown in proteins.[3]

Amino acid	3-Letter name	Symbol
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartic acid	Asp	D
Cysteine	Asx	B
Glutamine	Cys	C
Glutamic acid	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methiodine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

For clear understanding, a notation for HMM is summarized below. Furthermore, two concepts of tth state in perspective of time(sequence) and ith or jth state in a set of possible states are confusing in context, so 'step' denotes meaning of state in time dimension.

- $O = O_1, O_2, ..., O_{T-1}, O_T$
  : A sequence of observations(symbols). The sequence of a single protein is $O_1O_2O_3...O_{T-1}O_T$, for example, 'SIPPEV'.
- $S = S_1, S_2, ..., S_{T-1}, S_T$
  : A sequence of corresponding states. The linear set of secondary structures of a single protein is $S_1S_2S_3...S_{T-1}S_T$, for example, 'chheec'.
- $o = o(1), o(2), ..., o(m-1), o(m)$
  : A set of possible observations(symbols). Each symbol denotes a alphabet letter of amino acid, 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'Y', here $m = 20$.
- $s = s(1), s(2), ..., s(n-1), s(n)$
  : A set of possible states. Each hidden state denotes a secondary structure, 'h', 'e', 'c', here $n = 3$.
- π
  : An initial condition or start probability. Probability of a possible state at the first step of sequence.
- A
  : Transition probability. Probability of one state which transits to a specific state at next step. A set can be described as transition matrix with elements.

$$A = (a_{i,j}),\ i = 1-n, j = 1-n$$

  where $a_{i,j}$ is a probability of transition from $s(i)$ to $s(j)$
- B
  : Emission probability. Probability of one state which emits(displays) a specific

symbol(observation). A set can be described as emission matrix with elements.

$$B = (b_{i,j}),\ i = 1 - n, j = 1 - n$$

where $b_{i,j}$ is a probability of emission from $s(i)$ to $o(j)$

- $\lambda = (\pi, A, B)$
  : A hidden Markov Model with start probability, transition probability and emission probability.
- $\alpha_t(i)$
  : Forward probability. Probability of s_i hidden state at tth step when a sequence of observations is $O_1O_2...O_t$ with a given λ.
- $\beta_t(i)$
  : Backward probability. Probability of s_i state at tth step when a sequence of observations is $O_tO_{T-1}...O_T$ with a given λ.

As the λ is defined, following states and observations are determined with A and B. The probability of a hidden state is calculated from the probability of previous step and transition probability. The state at the first step is refered from the start probability, π and further steps are predicted by Markov chain property as below.

$$P(s_{t+1} = s(j)|s_t, \lambda) = \sum_{i=1}^{n} P(s_t(i)|\lambda) \times a_{i,j}$$

where s_t is a state at tth step. The probability of observed symbol is calculated from the sum of emission probabilities at a given step.

$$P(O(t) = o(j)|\lambda) = \sum_{i=1}^{n} P(s_t(i)|\lambda) \times b_{i,j}$$

where $O(t)$ is observation at tth step, $o(j)$ is jth observation(symbol).

Forward-backward algorithm

Forward probability and backward probability have similar concept with a probability of one ith state at given tth step, however, forward probability at tth step is computed from a sum of forward probabilites of the previous $(t-1)$th step, and backward probability is computed from a sum of hidden states at post $(t+1)$th step. As a sequence of observations $O = O_1O_2...O_T$ is given, the likelihood of tth step can be calculated from multiplying of forward probability(α) and backward probability(β). Each can be explained as below.

$$\alpha_t(j) = \sum_{i=1}^{n} \alpha_{t-1}(i) \times a_{ij} \times b_j(o_t)$$

$$\beta_t(i) = \sum_{j=1}^{n} a_{ij} \times b_j(o_{t+1}) \times \beta_{t+1}(j)$$

where $\alpha_{t-1}(i)$ is α of $s(i)$ at $(t-1)$th step, $b_j(o_t)$ is β from $s(j)$ to o_t and $\beta_{t+1}(j)$ is β of $s(j)$ at $(t+1)$th step.

and a likelihood at tth step is

$$P(O|\lambda) = \sum_{j=1}^{n} \alpha_t(j) \times \beta_t(j)$$

Viterbi algorithm

Viterbi algorithms is for prediction of a sequence of hidden states by finding maximum probability among states at a given step. It is basically similar with forward algorithm but the interest is focused on which state has the highest probability among possible states at a given step. The difference is more obvious with mathematical expression of 'max' comparing to 'sum' as below.

$$v_t(j) = \max_{i}^{n} [v_{t-1}(i) \times a_{ij} \times b_j(o_t)]$$

where $v_t(j)$ is a Viterbi probability of $s(j)$ state at tth step.

By finding the state which has the highest Viterbi probability value for each steps and we can find one sequence, S, and this process is called as prediction, or decoding. This part is mainly related to prediction of secondary structure of protein.

Baum-Welch algorithm

The importance of Forward-backward algorithm is basically at updating A and B with fixing model parameters. Simultaneous updating of model parameters are impossible, but implement of new parameters as ξ and γ can make it available with maximizing a likelihood of one sequence. This is called as Baum-Welch algorithm and also called as Expectation-Maximization(EM) algorithm.[4]

For updating α and β, we use new parameters γ and ξ. γ is for updating B and ξ is for updating A.

$$\gamma_t(j) = \frac{\alpha_t(j) \times \beta_t(j)}{\sum_{s=1}^{n} \alpha_t(s) \times \beta_t(s)}$$

$$\hat{b}_j(v_k) = \frac{\sum_{t=1, s.t. o_t = v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

$$\xi_t(i,j) = \frac{\alpha_t \times a_{ij} \times b_j(o_{t+1}) \times \beta_{t+1}(j)}{\sum_{s=1}^{n} \alpha_t(s) \times \beta_t(s)}$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{k=1}^{n} \xi_t(i,k)}$$

where T is a total number of amino acids in one sequence and k is same as n, the number of possible states. Especially, both expressions of ξ and γ refers Bayes theorem [5].

Implementation

Datasets and usage

A training set includes 111 proteins and secondary structure sequences and a testing set includes 17 proteins and secondary structure sequences. They are both provided from Terrence J. Sejnowsk [6], and this data sets are originally for studying prediction of secondary structure of globular proteins with Chou-Fasman algorithm.[7] The description of datasets are also included at refered link. [8]

Simple-HMM and EM-HMM

A training set is used in both simple-HMM and EM-HMM, however, simple-HMM only refers (π, A, B) from frequencies of states and symbols. For example, start probabilities of the simple-HMM is based on a method of counting frequencies of each state at the first step in sequences. Here, we denotes this method as simple counting method. In same way, transition probabilities are calculated by frequencies of sets of states from previous step to next step directly and emission probabilities are calculated by frequencies of symbols shown in each state. The difference between simple-HMM and EM-HMM is described in figure 2. Simple-HMM and EM-HMM both initialize by data from a training set, but Simple-HMM decodes by Vitebi algorithm and EM-HMM decodes by Viterbi algorithm with updating parameters by EM algorithm.

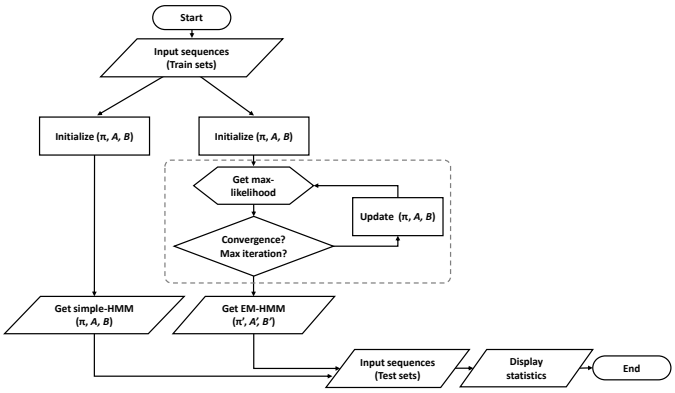

Figure 2 Block diagram of simple-HMM and EM-HMM The dashed area depicts EM-trained algorithm.

Table 2 Parameters for simple-HMM and EM-HMM

Parameter	simple-HMM	EM-HMM
Train population size	111	111
Test population size	17	17
length of protein in train sets	26-498	26-498
length of protein in test sets	35-461	35-461
Smoothing factor	0.00001	0.00001
Iteration	0	100/convergence

Program and modules

A structure of program for predicting protein secondary structure by simple-HMM and EM-HMM is implemented with Python 3.6 and Pandas, Math libraries. [9] This

program includes Statistics, Mathematics and Sequence modules. Statistics module is for analysing data for statistical tables. Mathematics module is for calculating simple probability or frequency. Sequence module is for reading raw input data and translating to sequence that main program can understand.

Accuracy

A measurement of accuracy is calculated as Q_3 for judging how the implemented method is qualified. It is described as a ratio of the number of correctly predicted secondary structure to the number of total secondary structure. Mathematical description is shown below,

$$Q_3(\%) = \frac{\sum_{n=1}^{N}\sum_{t=1}^{T} f(s_t, s_{t,test})}{\sum_{n=1}^{N}\sum_{t=1}^{T} f(s_t, s_t)} \times 100$$

$$f(i,j)\begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

where N is the number of proteins in sets of different implemented methods, for example, Simple Viterbi algorithm or EM algorithm. T is the number of states, that is the number of amino acids in one single protein. For investigating an affect of lengths of proteins and Q_3, Q_3 for a single protein is denoted as Q_{3s} and is described as a ratio of the number of correctly predicted secondary structure in a single protein to the number of a single protein. Different accuracy measurements depending on three secondary structure states are implemented. The Q_3 for the correctly predicted α-helix secondary structure is Q_{3a}. Likewise, Q_{3b} is for the correctly predicted β-sheet and Q_{3c} is for the correctly predicted random-coil.[10]

Iteration criteria

EM algorithm is used for updating parameters and the stopping criteria of iteration is needed for implementation. The first criteria is a maximum number of iteration, 100 here. Furthermore, EM algorithm estimates the logarithms of likelihoods, and a ratio of the previous likelihood to post likelihood is calculated at iteration. If the ratio is higher than 99% we assume convergence occurs, then iteration stops as described below. [4]

$$logP_i - logP_{i-1} < d$$

where a parameter $d = 0.0101$ and i is a iteration step.

Smoothing and scaling factor

For avoiding zero probability, we use one of smoothing methods called Laplace smoothing(additive smoothing)[11]. The smoothing parameter is $p = 0.00001$ for calculating π, A and B. Likewise, Python program can not distinguish very small number as $1e-320$ therefore scaling factor is used for calculating α and β. [12]

Result and Discussion

Comparison of simple Viterbi alogorithm and EM alogrithm

Prediction of protein secondary structure is expected to show higher accuracy at EM-HMM than simple-HMM, which is only based on Viterbi algorithm for decoding without EM-updating, however, comparing Q_3 of two models show an opposite results. 3 The first initialization of two models are same as the simple counting method so iterative EM algorithm is not as effective as expectation.

Table 3 Prediction results from simple-HMM method, EM-HMM method with 100 iteration and EM-HMM method with convergence.

Model/Accuracy(%)	Q_3	Q_{3a}	Q_{3b}	Q_{3c}
Simple	54.6	0	0	100
EM iteration	42.4	2.4	41.3	60.5
EM convergence	37.8	49.0	5.7	45.4

Furthermore, the frequency result in table 4 shows all predicted structures are random-coils in simple-HMM. This is quite useless prediction because simple-HMM model shows only one secondary structures prediction for all proteins. Therefore, comparison of Q_{3a}, Q_{3b} and Q_{3c} is more valuable for qualifying these three models. Moreover, the predicted raw data also shows all amino acids are in random-coil state in simple-HMM method.

Simple-HMM shows 0% for Q_{3a}, Q_{3b} but EM-HMM with iteration or convergence both show appropriate accuracy result. 3 This is basically because convergence means smaller iteration number than 100(from 7 to 9 as estimated) and increasing iteration number can increase accuracy and improve this program. Moreover, we can infer that the transition probability is adjusted with increasing iteration number by two columns of coils from original states to predicted states comparing EM iteration and EM convergence in table 4.

Table 4 Frequency table of simple-HMM method, EM-HMM method with 100 iteration and EM-HMM method with convergence. Predict shows the number of predicted states by each model and orig shows the number of states in original test set.

Model	Simple			EM iter.			EM conv.		
Predict/orig	c	e	h	c	e	h	c	e	h
Coil(c)	1923	748	849	1164	418	570	1317	785	48
Sheet(e)	0	0	0	735	309	259	784	489	15
Helix(h)	0	0	0	24	21	20	39	24	2

One possible assumption of difference between this suspicious simple-HMM and reasonable EM-HMM result from both models can be the process of updating affects the result or not. The transition probabilities of two models(simple-HMM and EM-HMM iteration) are shown in figure 3. The interesting point here is two plots show similarity at most of amino acid cases except D(Aspartic acid), G(Glycine), L(Leucine) and especially the transition probability of leucine is obviously different. This can be proof that a special structure including leucine can affect the prediction of secondary structure. It can be a leucine zipper, as an exception of this prediction algorithm. [13] Moreover, other structures such as zinc finger can also interfere methods of prediction used here.

Combining BLAST algorithm with the algorithm mentioned here can be one of options for improving the shortage. In PSI-BLAST algorithm, decoding is especially specified at the confined region. Therefore, the confined sequence region which has

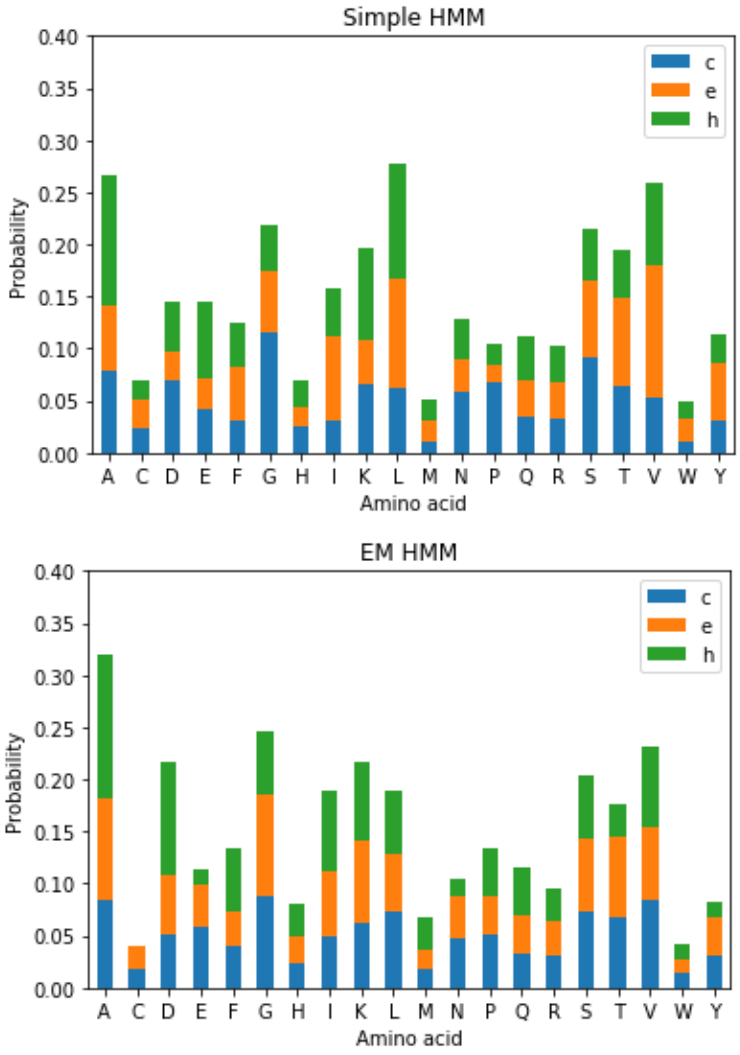

Figure 3 Transition probability of simple EM model and EM iteration model. Simple-HMM(above) and EM-HMM iter.(below)

obviously different from others do not affect successive decoding process at other sequence region. In EM-HMM, this region can affect transition probability as updating and it can result the lower accuracy of further decoding processes.[14]

Other existent methods show higher accuracy, such as block HMM shows 58-76% [15] and PSI-BLAST shows 55-94.7%.[1] Furthermore, these other methods show that the length of protein is also parameter which can affect the accuracy and further study of our program. This part should be implemented with comparing single-protein and multiple-protein validation.

Conclusion

HMM implemented here shows high accuracy and proves its benefit. The simple counting method can predict protein secondary structure with high accuracy, however, HMM with EM algorithms is more beneficial to predict protein secondary

structure in perspective of Q_{3a}, Q_{3b}, Q_{3c}. Comparing to other methods such as corresponding BLAST algorithm, EM-HMM method here shows not high accuracy but can be potential as iteration increases the accuracy. [16, 1]

Availability and requirements
Related program codes and files can be downloaded from link below
https://github.com/hyejeonc/proteinhmm [8]

References

1. Jones, D.T.: Protein secondary structure prediction based on position-specific scoring matrices. Journal of molecular biology **292**(2), 195–202 (1999)
2. Example Diagram of HMM. http://yanfenglu.net/researchVAS$_p1.htm$
3. Abela, J., Michael, J.: Topics in evolving transformation systems [microform]. (2019)
4. Sheh, A., Ellis, D.P.: Chord segmentation and recognition using em-trained hidden markov models (2003)
5. Bayes Theorem. https://www.britannica.com/topic/Bayess-theorem
6. Hidden Markov Models. https://archive.ics.uci.edu/ml/datasets/Molecular+Biology+(Protein+Secondary+Structure)
7. Qian, N., Sejnowski, T.J.: Predicting the secondary structure of globular proteins using neural network models. Journal of molecular biology **202**(4), 865–884 (1988)
8. Main Program Repository of Author. https://github.com/hyejeonc
9. Matlib Library. https://matplotlib.org/
10. Asai, K., Hayamizu, S., Handa, K.: Prediction of protein secondary structure by the hidden markov model. Bioinformatics **9**(2), 141–146 (1993)
11. Boodidhi, S.: Using smoothing techniques to improve the performance of hidden markov's model (2011)
12. Shifrin, J., Pardo, B., Meek, C., Birmingham, W.: Hmm-based musical query retrieval. In: Proceedings of the 2nd ACM/IEEE-CS Joint Conference on Digital Libraries, pp. 295–300 (2002). ACM
13. Landschulz, W.H., Johnson, P.F., McKnight, S.L.: The leucine zipper: a hypothetical structure common to a new class of dna binding proteins. Science **240**(4860), 1759–1764 (1988)
14. Söding, J.: Protein homology detection by hmm–hmm comparison. Bioinformatics **21**(7), 951–960 (2004)
15. Won, K.-J., Hamelryck, T., Prügel-Bennett, A., Krogh, A.: An evolutionary method for learning hmm structure: prediction of protein secondary structure. BMC bioinformatics **8**(1), 357 (2007)
16. Lee, L., Leopold, J.L., Frank, R.L.: Protein secondary structure prediction using blast and relaxed threshold rule induction from coverings. In: 2011 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), pp. 1–8 (2011). IEEE

Additional Files
main code — hiddenmarkovmodel.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar  4 17:25:48 2019

@author: HYEJEONG

"""
from mathematics import *
import sequence as seq
import statistics as stat

import pandas as pd
from math import log #Problem with float --> change to log

class Hmm(object):
    """
    Main Class of Hiddem Markov Model.
    Hiddem Markov Model(lambda) is specified with three parameters
    Pi : prob_start, start probability
    A : prob_trans, transition probability
    B : prob_emit, emission probability
    State list and symbol list are also included.
    """
    def __init__(self, prob_start, prob_trans, prob_emit, statelist, symbollist):
        #Define model with five parameters
        self._prob_start = prob_start
        self._prob_trans = prob_trans
        self._prob_emit = prob_emit
        self._statelist = list(statelist)
        self._symbollist = list(symbollist)
```

```
def get(self):
    #Show parameters
    return self._prob_start, self._prob_trans, self._prob_emit, self._statelist, self._symbollist

def zero(self):
    #Initialize to all probabilites to 0.0
    self._prob_trans = {}
    self._prob_emit = {}
    self._prob_start = {}

    for state in self._statelist:
        self._prob_trans[state] = {}
        self._prob_emit[state] = {}
        for post_state in self._statelist:
            self.prob_trans[state][post_state] = 0.0
        for symbol in self._symbollist:
            self._prob_emit[state][symbol] = 0.0
        self._prob_start[state] = 0.0

def check(self, sequence):
    #Check the sequence is appropriate
    if sequence not in self._symbollist:
        print('Not available sequence')

def prob_start(self):
    #Show start probability
    if self._prob_start == None:
        return 0
    return self._prob_start

def prob_trans(self):
    #Show transition probability
    if self._prob_trans == None:
        return 0
    return self._prob_trans

def prob_emit(self):
    #Show emission probability
    if self._prob_emit == None:
        return 0
    return self._prob_emit

def statelist(self):
    #Show state list
    if self._statelist == None:
        return None
    return self._statelist

def symbollist(self):
    #Show symbol list
    if self._symbollist == None:
        return None
    return self._symbollist

def emupdate(self, sequence):  #For updating, Baum-Welch(Forward-Backward algorithm)
    """
    This is a method for updating A, B by calculating value with A*B
    It is based on Expectation-Maximization algorithm
    """
    a = self.forward(sequence)
    b = self.backward(sequence)

    # Expectation value (E-step)
    g = [] # get gamma
    g_smooth = []
    sum_prob = 0.0
    for t in range(0, len(sequence)): # 0, 1, ... l-1 step
        g.append({})
        for state in self._statelist:
            g[t][state] = a[t][state] * b[t][state]
            sum_prob += g[t][state]
```

```
            g_smooth.append(sum_prob)
            if sum_prob > 1e-300:
                for state in self._statelist:
                    g[t][state] /= sum_prob

        x = [] # get xi
        x_smooth = []
        for t in range(0, len(sequence)-1): # 0, 1, ... l-2 step
            x.append({})
            sum_prob = 0.0
            for pre_state in self._statelist:
                x[t][pre_state] = {}
                for post_state in self._statelist:
                    x[t][pre_state][post_state] = a[t][pre_state]  \
                        * self._prob_trans[pre_state][post_state] \
                        * self._prob_emit[post_state][sequence[t+1]] \
                        * b[t+1][post_state]
                    sum_prob += x[t][pre_state][post_state]
                x_smooth.append(sum_prob)

            if sum_prob > 1e-300:
                for pre_state in self._statelist:
                    for post_state in self._statelist:
                        x[t][pre_state][post_state] /= sum_prob


        #Maximization step (M-step)
        #Laplace smoothing
        p = 0.00001
        for state in self._statelist:
            #for start probabilities
            self._prob_start[state] = (g[0][state] + p) / (1 + p * len(self._statelist))

            #for transition probabilities
            for post_xstate in self._statelist:
                sum_sum_prob = 0.0
                sum_x = 0.0
                for t in range(0, len(sequence)-1): #fraction denominator
                    sum_prob= 0.0
                    for post_state in self._statelist:
                        sum_prob += x[t][state][post_state]
                    sum_sum_prob += sum_prob

                    sum_x += x[t][state][post_xstate] #fraction numerator

                    self._prob_trans[state][post_xstate] = (sum_x + p) \
                    / (sum_sum_prob + p*len(self._statelist))

            #for emission probabilities
            sum_g = 0.0
            for t in range(0, len(sequence)):
                sum_g += g[t][state]
            sum_g += g[len(sequence)-1][state]

            sum_g_emit = {}
            for symbol in self._symbollist:
                sum_g_emit[symbol] = 0.0

            for t in range(0, len(sequence)):
                sum_g_emit[sequence[t]] += g[t][state]

            for symbol in self._symbollist:
                self._prob_emit[state][symbol] = (sum_g_emit[symbol] + p) \
                / (sum_g + p*len(self._statelist))
        return self._prob_emit, self._prob_trans

    def forward(self, sequence):
        """
        This is a method for calculating forward probability.
        output must be a list as below.
        a = [ {'h':... , 'e':... , '_':... },    a_{0}
                            ...
```

```
            {'h':... , 'e':... , '_':... }, ]  a_{length-1}
        """
        a = []   # forward probability, alpha (list for [t steps];
                 # dict for probabilities of states {'h', 'e', '_'})
                 # alpha is saved for next alpha, dynamics programming
        c = [{}] # scaling factor for very very small number

        for t in range(0, len(sequence), +1): # 0, 1, ... , l-1 step
            a.append({})
            if t == 0:
                for state in self._statelist:

                    a[t][state] = self._prob_start[state] * self._prob_emit[state][sequence[0]]
            else:
                for state in self._statelist:

                    sum_prob = 0.0
                    for pre_state in self._statelist:
                        sum_prob += a[t-1][pre_state] * self._prob_trans[pre_state][state]
                    a[t][state] = sum_prob * self._prob_emit[state][sequence[t-1]]

            if sum(list(a[t].values())) > 1e-300:
                c = 1 / sum(list(a[t].values()))
                for state in self._statelist:
                    a[t][state] *= c

        return a

    def backward(self, sequence):
        """
        This is a method for calculating backward probability.
        Contrary, b is calculated from the last step, so b_{0} is calculated and stacked further.
        Output must be a list as below.
        b = [ {'h':... , 'e':... , '_':... },    b_{0}
                              ...
              {'h':... , 'e':... , '_':... }, ]  b_{length-1}
        """
        b = [] # backward probability, beta (list for [t states];
               # dict for probabilities of states {'h', 'e', '_'})
               # beta is saved for next beta, dynamics programming
        c = [] # scaling factor for very very very small number
        for t in range(len(sequence)-1, -1, -1): # 0, 1, ... , l-1 step
            b.insert(0, {})
            for state in self._statelist:
                if t == (len(sequence)-1):
                    b[0][state] = 1.0
                else:
                    sum_prob = 0.0
                    for post_state in self._statelist:
                        sum_prob += b[1][post_state] * self._prob_trans[state][post_state]

                    b[0][state] = sum_prob * self._prob_emit[state][sequence[t]]

            if sum(list(b[0].values())) > 1e-300:
                c = 1 / sum(list(b[0].values()))
                for state in self._statelist:
                    b[0][state] *= c
        return b

    def viterbi(self, sequence):
        """
        This is a method for Viterbi algorithm to decode.
        Output must be a list as below .
        v = [ {'h':... , 'e':... , '_':... },    v_{0}
                              ...
              {'h':... , 'e':... , '_':... }, ]  v_{length-1}
        """
        v = []
        dec_state = []
        dec_prob = []
        for t in range(0, len(sequence)): # state number : 0, 1, ... , l-1 step
            v.append({})
```

```
            if t == 0:
                for state in self._statelist:
                    v[t][state] = self._prob_start[state] * self._prob_emit[state][sequence[0]]
            else:
                for state in self._statelist:
                    v[t][state] = vmax[0] * self._prob_trans[vmax[1]][state] \
                    * self._prob_emit[state][sequence[t-1]]

            vmax = list(max(zip(v[t].values(), v[t].keys())))

            if vmax[0] > 1e-300 :
                c = 1 / sum(list(v[t].values()))
                for state in self._statelist:
                   v[t][state] *= c
                vmax[0] *= c

            dec_prob.append(vmax[0])
            dec_state.append(vmax[1])

        return [v[t], dec_state, vmax[0]]

    def initialize(self, proteinset):
        #Initialize a model by parameters from outside of class.
        #This is because we can not used the model parameter by simple counting method after training
        self._prob_start = initialset(proteinset)[1][0]
        self._prob_trans = initialset(proteinset)[1][1]
        self._prob_emit = initialset(proteinset)[1][2]
        self._statelist = initialset(proteinset)[2][0]
        self._symbollist = initialset(proteinset)[2][1]
        return prob_start, prob_trans, prob_emit, statelist, symbollist

    def train(self, initialset, trainset, initial=1, iteration = False, d=0.0101, maxiter=100):
        """
        This is a method for updating parameter by EM algorithm, related to stop iteration.
        Stopping criteria follows the log-likelihood.
        Parameters    |    Description
                      |
        initial       | Initializing method/dataset. Default = 1(set that already modeled)
        iteration     | Stopping criteria. Default = False(maxiter is not used.)
        d             | Stopping criteria for convergence. Default = 0.0101.
        maxiter       | Stopping criteria for iteration. Default = 100.
        """
        if initial == 0:
            self.zero()
        elif initial == 1:
            pass
        elif initial == 2:
            self.initialize(trainset)
        elif initial ==3 :
            self.initialize(initialset)

        proteins = trainset[0]
        structures = trainset[1]

        pre_prob = 0.0
        for protein in proteins:
            prob = self.viterbi(protein)
            pre_prob += log(prob[2]) #expectation
        pre_prob /= len(proteins)
        print(pre_prob/len(proteins))

        for i in range(maxiter):
            post_prob = 0.0
            print('Iteration number is .... ', i+1)
            for protein in proteins:
                #E-M update transition probabilites and emission probabilities
                self.emupdate(protein)
                prob = self.viterbi(protein) #viterbi probability
                post_prob += log(prob[2])
            post_prob /= len(proteins)
            print(pre_prob, post_prob)
```

```
            if iteration == False:
                if (post_prob > pre_prob) and (abs(post_prob - pre_prob) < d):
                    break
            #if post_prob <= pre_prob:
            pre_prob = post_prob

        return i, pre_prob/len(proteins), post_prob/len(proteins)

    def decode(self, proteinset):
        """
        This is a method for decoding(translating) by Viterbi algorithm.
        To find the  sequence that has highest probability.
        """
        proteins = proteinset[0]
        structures = proteinset[1]

        decode = []
        likelihood = []

        for protein, structure in zip(proteins, structures):
            viterbi = self.viterbi(protein)
            decode.append(viterbi[1])
            likelihood.append(viterbi[2])

        decodeset = [proteins, decode]
        return decodeset

def initialset(proteinset):
    """
    This is a method for finding parameters by raw data sets.
    Raw data sets are seperated as below.

    """
    proteins = proteinset[0]
    structures = proteinset[1]

    symbolcount = {}
    statecount = {}

    startcount = {}
    transcount = {}
    statesymbolcount = {}

    for protein, structure in zip(proteins, structures):
        pre_state = {}
        count1d(startcount, structure[0])
        for symbol, state in zip(protein, structure): #single protein
            count1d(statecount, state)
            count1d(symbolcount, symbol)
            count2d(statesymbolcount, state, symbol) # for transmission probability
            if pre_state != {}:
                count2d(transcount, pre_state, state) # for emittance probability
            pre_state = state

    statelist = list(statecount.keys())
    symbollist = list(symbolcount.keys())

    prob_trans = norm2d(transcount, statelist, statelist)
    prob_emit = norm2d(statesymbolcount, statelist, symbollist)
    prob_start = norm1d(startcount, statelist)
    #print(prob_start)
    return [startcount, transcount, statesymbolcount], [prob_start, prob_trans, prob_emit], \
           [statelist, symbollist]

def initialHmm(proteinset):
    prob_start = initialset(proteinset)[1][0]
    prob_trans = initialset(proteinset)[1][1]
    prob_emit = initialset(proteinset)[1][2]
    statelist = initialset(proteinset)[2][0]
    symbollist = initialset(proteinset)[2][1]
    return Hmm(prob_start, prob_trans, prob_emit, statelist, symbollist )
"""
```

```
def varname(p): #https://stackoverflow.com/questions/592746/how-can-you-print-a-variable-name-in-python
  for line in inspect.getframeinfo(inspect.currentframe().f_back)[3]:
    m = re.search(r'\bvarname\s*\(\s*([A-Za-z_][A-Za-z0-9_]*)\s*\)', line)
    if m:
      return m.group(1)
"""
def printdata(file, data):
    print(data)
    with open(file, "w") as f:
        f.write("\n\n######result of decoded sequence#####\n")
       # f.write(varname(data))
        f.write("\n##################################################\n")
        f.write("\n")
        f.write(str(data))
        f.write("\n")
        size = list(map(len, data[0]))
        for i in range(len(data[0])):
            f.write(str(i+1))
            f.write(str(data[0][i]))
            f.write("\n")
            f.write(str(data[1][i]))
            f.write("\n\n\n")
        pd.set_option('display.max_colwidth', 3*max(size))
        pd.set_option('display.max_rows', 3*max(size))
        tr = pd.DataFrame(data).T
        f.write(tr.to_string())

if __name__ == "__main__":
    path_train = r".\dataset\protein-secondary-structure.train"
    path_test = r".\dataset\protein-secondary-structure.test"
    path_decode = r".\output\raw_data.csv"
    rawlines = None
    while True:
        try:
            path_train = input('Insert input file for training \n(for example: .\dataset\protein-secondary-stru
            rawlines = seq.lineseq(path_train)
            #print(rawlines)
            if rawlines == None:
                print('Not available input file. Select options with integer number below (1, 2, 3).', \
                '1. Try another file', \
                '2. Use example input file (.\dataset\protein-secondary-structure.train)', \
                '3. Exit : Ctrl + C', \
                sep='\n')
                trytype = input()
                if trytype == 1: continue
                elif trytype == 2:
                    path_train = r'.\dataset\protein-secondary-structure.train'
                    break
                else:
                    print('Incorrect value. Try again')
                    continue
            else:
                break
        except FileNotFoundError or NameError:
            print('Not available input file. Select options with integer number below (1, 2, 3).', \
                '1. Try another file', \
                '2. Use example input file (.\dataset\protein-secondary-structure.train)', \
                '3. Exit : Ctrl + C', \
                sep='\n')
            trytype = input()
            if trytype == 1: continue
            elif trytype == 2:
                path_train = r'.\dataset\protein-secondary-structure.train'
                break
            else:
                print('Incorrect value. Try again')
                continue
    rawlines = None
    while True:
        try:
            path_test= input('Insert input file for testing \n(for example: .\dataset\protein-secondary-structu
            rawlines = seq.lineseq(path_test)
```

```
            if rawlines == None:
                print('Not available input file. Select options with integer number below (1, 2, 3).', \
                '1. Try another file', \
                '2. Use example input file (.\dataset\protein-secondary-structure.test)', \
                '3. Exit : Ctrl + C', \
                sep='\n')
                trytype = input()
                if trytype == 1: continue
                elif trytype == 2:
                    path_test = r'.\dataset\protein-secondary-structure.test'
                    break
                else:
                    print('Incorrect value. Try again')
                    continue
            else:
                break
        except FileNotFoundError or NameError:
            print('Not available input file. Select options with integer number below (1, 2, 3).', \
                '1. Try another file', \
                '2. Use example input file (.\dataset\protein-secondary-structure.test)', \
                '3. Exit : Ctrl + C', \
                sep='\n')
            trytype = input()
            if trytype == 1: continue
            elif trytype == 2:
                path_test = r'.\dataset\protein-secondary-structure.test'
                break
            else:
                print('Incorrect value. Try again')
                continue

    print(path_train, "is a file for training\n")
    print(path_test, "is a file for testing\n\n")
    print("decoded raw data is saved as")
    print(".\output\raw_data_method.csv")

    trainset = seq.getproteinset(path_train)
    testset = seq.getproteinset(path_test)

    trainmodel = initialHmm(trainset)
    data_simple = trainmodel.decode(testset)

    trainmodel_conv = initialHmm(trainset)
    trainmodel_conv.train(trainset, trainset)
    data_conv = trainmodel_conv.decode(testset)

    trainmodel_iter = initialHmm(trainset)
    trainmodel_iter.train(trainset, trainset, iteration=True)
    data_iter = trainmodel_iter.decode(testset)

    printdata(r'.\output\raw_data_simpleHMM.csv', data_simple)
    printdata(r'.\output\raw_data_EMHMMconv.csv', data_conv)
    printdata(r'.\output\raw_data_EMHMMiter.csv', data_iter)
    """
    Want more statistical analysis?
    printdata('summary5.csv', stat.summary(data_simple))
    printdata('summary4.csv', stat.summary(data_conv))
    printdata('summary4.csv', stat.summary(data_iter))

    printdata('accuracy5.csv', stat.accuracy(testset, data_simple))
    printdata('accuracy5.csv', stat.accuracy(testset, data_conv))
    printdata('accuracy5.csv', stat.accuracy(testset, data_iter))
    """
    """
    for curiosity, this is start from zero probability for all parameters, (lambda, A, B)

    with open("Result_comparetrains.csv", "a") as f:
        trainmodel.train(trainset, trainset, initial=0)
        f.write("\n\n#######EM trained result - Convergence / from initial \n\n")
        f.write(str(trainmodel.decode(testset)))
        f.write("\n\n")
    T = pd.DataFrame(trainmodel.decode(testset)).T
```

```
    T.to_csv("Result_comparetrains.csv", mode='a', header=True)  # Save in result.txt file

    with open("Result_comparetrains.csv", "a") as f:
        trainmodel.train(trainset, trainset, initial=0, iteration=True)
        f.write("\n\n#######EM trained result - maxiter / from initial \n\n")
        f.write(trainmodel.decode(testset))
        f.write("\n\n")
    T = pd.DataFrame(trainmodel.decode(testset)).T
    T.to_csv("Result_comparetrains.csv", mode='a', header=True)  # Save in result.txt file
    """
```

module — sequence.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Feb 13 16:49:57 2019

@author: HYEJEONG
"""
from collections import Counter
from mathematics import *

#aminoacid = ['A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'Y
#structure = ['h', 'e', '_']


#symbollist = {'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', '
#statelist = {'h', 'e', '_'}

def readfile(string):
        """
        This is a method to determine if argument is a string representing a numeric value.
        """
        for kind in (str, str, str):
            try:
                kind(string)
            except (TypeError, ValueError):
                pass
            else:
                return True
        else:
            return False

def lineseq(path):
    """
    This method is for seperating string to word for getting symbols and states.
    """
    allstring = []
    with open(path) as f:
        for line in (line.strip() for line in f):  ## line                    split
         fields = line.split()
         if fields: # non-blank line?
             if readfile(fields[0]):
                 allstring += fields
    return allstring

def proteinseq(path, dbtype = 1):
    """
    This is a method for getting a protein set.
    output = [ [ [ protein1 ], [ protein2 ], [ protein3 ], [ protein4 ], ... ],
               [ [ structure1 ], [ structure2 ], [ structure3 ], [ structure4 ], ... ] ]
    """
    if dbtype == 0:
        None #FASTA file reading method will be here... To be continued...
    else:
        allstring = lineseq(path)
        protein = []
        secondstr = []
        i = None
        for j in range(len(allstring)):
            if allstring[j] == '<>' or allstring[j] == '>':
```

```
                protein_single = []
                secondstr_single = []
                i = j+1
                while i < len(allstring):
                    if allstring[i] == 'end' or allstring[i] == '<>' or allstring[i] == '<end>'
:
                        protein.append(protein_single)
                        secondstr.append(secondstr_single)
                        protein_single = []
                        secondstr_single = []
                        break
                    else: #allstring[i] != '<' or allstring[i] != '>' or allstring[i] != '<>':
                        protein_single.extend(allstring[i])
                        secondstr_single.extend(allstring[i+1])
                        i += 2
    return protein, secondstr

def getproteinset(path):
    proteinset = proteinseq(path)
    return proteinset
```

module — mathematics.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 19 10:23:37 2019

@author: HYEJEONG
"""
#This module has methods that is simply for mathematical usage.


def count1d(count, item): #Count +1 'count' if 'item' is in 'count'.
    if item not in count:
        count[item] = 0
    count[item] += 1

def count2d(count, item1, item2): #Count +1 'count' if 'item' is in 'count', for 2-dimensional array(dictionary
    if item1 not in count:
        count[item1] = {}
    count1d(count[item1], item2)

def norm1d(prob, item_set):  #Normalize 1-dimensional array for getting probability
    result = {}
    prob_sum = 0.0

    for item in item_set:
        prob_sum += prob.get(item, 0)

    if prob_sum == 0:
        result[item] = 0
    else:
        for item in item_set:
            result[item] = prob.get(item, 0) / prob_sum
    return result

def norm2d(prob, item_set1, item_set2): #Normalize 2-dimensional array for getting probability
    result = {}

    if prob is None:
        for item in item_set1:
            result[item] = norm1d(None, item_set2)
    for item in item_set1:
        result[item] = norm1d( prob.get(item, 0), item_set2)

    return result
```