Ukryte Łańcuchy Markova

Szymon Kuliński, Piotr Głownia, Jakub Jurczak, Janusz Utrata
Bartek Sroczyński

Contents

1 Wstęp **1**

2 Teoria **2**
2.1 Łańcuchy Markova 2
2.2 Dyskretne ukryte procesy Markova (ang. HMM) 3
2.3 Algorytm Forward-Backward 4
2.3.1 Problem 5
2.3.2 Inny algorytm 5
2.3.3 Inicjalizacja 5
2.3.4 Rekurencja 6
2.3.5 Prawdopodobieństwo 6
2.3.6 Zmienna do-tyłu 6
2.4 Algorytm Viterbi 7
2.4.1 Zagadnienie 7
2.4.2 Reformulacja problemu 7
2.4.3 Inne spojrzenie na algorytm 7
2.5 Odległości między HMM 8

3 Implementacja **9**
3.1 Implementacja błądzenia losowego w Pythonie 9
3.2 Implementacja HMM C++ 12

4 Wyniki i podsumowanie **21**

1 Wstęp

Łańcuch Markova to automat skończony z prawdopodobieństwami dla każdego przejścia, tzn. prawdopodobieństwo, że następny stan wynosi s_j poprzez dany obecny stan s_i.
Równoważnie: Łańcuch Markova jest opisywany poprzez graf kierunkowy z pewnymi wagami w którym to grafie wagi są powiązane z prawdopodobieństwem przejścia na następny stan. Wagi są nieujemne i suma wag wychodzących krawędzi jest dodatnia. Jeżeli wagi są znormalizowane całkowita waga wynosi 1.
Ukryty model Markova to automat skończony z prawdopodobieństwami dla każdego przejścia tzn. prawdopodobieństwo, że następny stan wynosi s_j poprez dany obecny stan s_i. Stany nie

są bezpośrednio obserwowane. W zamian, każdy stan produkuje jeden z możliwych do zaobserwowania wyników z pewnym prawdopodobieństwem.
Obliczenie modelu poprzez dane zbiory sekwencji zaobserwowanych wyników jest trudne, stany nie są bezpośrednio obserwowalne, a przejścia są probabilistyczne. Chociaż stany nie mogą być zgodnie z definicja bezpośrednio obserwowalne, najbardziej prawdopodobna sekwencja zbiorów dla danej sekwencji obserwowanych wyników może być obliczona w czasie $O(nT)$ gdzie n to liczba stanów a T to długość sekwencji.
Ukryty proces Markova to łańcuch Markova, gdzie każdy stan generuje obserwację. Widzimy tylko obserwacje, a celem jest wywnioskować sekwencję stanu ukrytego.

2 Teoria

2.1 Łańcuchy Markova

Def. Wektor $\mathbf{p} = (\mathbf{p_0}, \mathbf{p_1}, ..., \mathbf{p_{N-1}})$ nazywamy wektorem prawdopodobieństwa jeśli składowe są nieujemne i ich suma wynosi 1.

$$\sum_{i=0}^{N-1} \mathbf{p_i} = \mathbf{1}$$

Def. Macierz kwadratowa NxN $A = (a_{ij})$ jest nazywana stochastyczną jeśli jej wiersze tworzą wektory prawdopodobieństwa.

Def. Macierz stochastyczna A jest zwaną regularną jeśli wszystkie jej elementy po podniesieniu jej do jakiejś potęgi A^n są dodatnie.

Def. Punkt stały $\mathbf{q}$ regularnej macierzy stochastycznej A jest definiowany jako:

$$\mathbf{qA} = \mathbf{q}$$

Tw. Niech A będzie regularną macierzą stochastyczną. Wtedy:

(i) A posiada jednoznaczny skończony wektor prawdopodobieństwa $\mathbf{q}$ a jego składowe są dodatnie.

(ii) ciąg macierzy $A, A^2, A^3, ...$ potęg A zbiega do macierzy B której każdy wiersz stanowi punkt stały $\mathbf{q}$.

(iii) jeżeli $\mathbf{p}$ jest dowolnym wektorem prawdopodobieństwa, to ciąg wektorów $\mathbf{p}A$, $\mathbf{p}A^2$, $\mathbf{p}A^3$, ... zbiega do punktu stałego $\mathbf{q}$.

$$o_0, o_1, o_2, ..., o_{T-1}$$

spełniają dwie własności:

(i) Każdy rezultat należy do skończonego zbioru rezultatów:

$$\{q_0, q_1, q_2, ..., q_{N-1}\}$$

zwanego przestrzenią stanową układu. Jeżeli rezultat przy t-ej próbie wynosi q_i, mówimy, że układ jest w stanie q_i w czasie t lub w t-ym kroku.

(ii) Rezultat dowolnej próby zależy co najwyżej od rezultatu najbliższej poprzedniej próby, nie zależąc od poprzednich prób, z każdą parą stanów (q_i, q_j) dane jest prawdopodobieństwo a_{ij}, że q_j wydarzy się zaraz po wydarzeniu q_i.

Taki proces stochastyczny nazywany jest skończonym łańcuchem Markova. Liczy a_{ij}, zwane prawdopodobieństwami przejścia mogą być ułożone w macierz kwadratową NxN:

$$\begin{pmatrix} a_{00} & a_{01} & ... & a_{0N-1} \\ a_{10} & a_{11} & ... & a_{1N-1} \\ ... & ... & \ddots & ... \\ a_{N-10} & a_{N-11} & ... & a_{N-1N-1} \end{pmatrix}$$

zwaną macierzą przejścia. Macierz przejścia A łańcuchu Markova jest macierzą stochastyczną.

2.2 Dyskretne ukryte procesy Markova (ang. HMM)

Używamy następującej notacji:
N - licza stanów w modelu
M - całkowita liczba różnych obserwacji symboli alfabetu. Jeżeli obserwacje są ciągłe to M jest nieskończone. Zakładamy, że M jest skończone.
T - długość sekwencji obserwacji (zestaw danych do wytrenowania) gdzie:
t = 0, 1, ..., T - 1. Istnieje zatem N^T możliwych sekwencji stanu. Niech $(\Omega_q := q_0, q_1, ..., q_{N-1})$ będzie skończonym zbiorem możliwych stanów.
Niech $(V := v_0, v_1, ..., v_{M-1})$ będzie skończonym zbiorem możliwych symboli obserwacji.
q_t to zmienna losowa oznaczająca stan w czasie t (zmienna stanu)
o_t to zmienna losowa oznaczająca obserwację w czasie t (zmienna wynikowa)
$O = (o_0, o_1, ..., o_{T-1}$ to sekwencja rzeczywistych obserwacji.
Zbiór prawdopodobieństw dla przejść stanu to $A = (a_{ij})$ gdzie $i, j = 0, 1, ..., N-1$
$a_{ij} = p(q_{t+1} = j | q_t = i)$ gdzie p jest prawdopodobieństwem stan-przejście tzn. prawdopodobieństwem bycia w stanie j w czasie t+1 gdy układ był w stanie i w czasie t. Zakładamy, że a_{ij}-ty są niezależne od czasu t.

Mamy więc warunki dla a_{ij}:

$$a_{ij} \geq 0 \; dla \; i, j \; = 0, 1, ..., N-1$$

$$\sum_{j=0}^{N-1} a_{ij} = 1 \; dla \; i, \; j = 0, 1, ..., N-1$$

Więc A jest macierzą $N \times N$

Przykład. Jeżeli mamy 6 stanów macierz przejścia w rozpoznawaniu mowy może wyglądać następująco:

$$A = \begin{pmatrix} 0.3 & 0.5 & 0.2 & 0 & 0 & 0 \\ 0 & 0.4 & 0.3 & 0.3 & 0 & 0 \\ 0 & 0 & 0.4 & 0.2 & 0.4 & 0 \\ 0 & 0 & 0 & 0.7 & 0.2 & 0.1 \\ 0 & 0 & 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 1.0 \end{pmatrix}$$

Warunkowy rozkład prawdopodobieństwa obserwacji w t, o_t, w danym stanie j wynosi:
$b_j(k) = p(o_t = v_k | q_t = j)$ gdzie $j \in \{0, 1, ..., N-1\}$ a $k \in \{0, 1, ..., M-1\}$ tzn. p jest prawdopodobieństwem zaobserwowania symbolu v_k poprzez dany stan j. Niech $B := \{b_j(k)\}$.
Mamy wtedy:

$$b_j(k) \geq 0$$

dla $j \in \{0, 1, ..., N-1\}$ i $k \in \{0, 1, M-1\}$ oraz:

$$\sum_{k=0}^{M-1} b_j(k) = 1$$

dla $j = 0, 1, ..., N-1$
Rozkład stanu początkowego $\pi = \{\pi_i\}$:

$$\pi_i = p(q_0 = i)$$

gdzie $i = 0, 1, ..., N-1$ tj. π_i jest prawdopodobieństwem bycia w stanie i w $t = 0$
Możemy teraz jasno zdefiniować ukryty model Markova (HMM z ang. Hidden Markov Model).
HMM to lista 5 elementowa:

$$(\Omega_q, V, A, B, \pi)$$

2.3 Algorytm Forward-Backward

Dany jest model $\lambda = (A, B, \pi)$. Jak możemy policyć $P(O|\lambda)$, prawdopodobieństwo wystąpienia sekwencji obserwacji:

$$O = (o_0, o_1, ..., o_{T-1})$$

Metoda bezpośrednia polega na znalezieniu $P(O|\mathbf{q}, \lambda)$ dla sekwencji stanu skończonego. Wtedy mnożymy przez $P(\mathbf{q}|\lambda)$ i sumujemy przez wszystkie możliwe sekwencje stanu. Mamy:

$$P(O|\mathbf{q}, \lambda) = \prod_{t=0}^{T-1} P(o_t | q_t, \lambda)$$

gdzie założyliśmy statystyczną niezależność obserwacji. Znajdujemy więc:

$$P(O|\mathbf{q},\lambda) = \prod_{t=0}^{T-1} P(o_t|q_t,\lambda)$$

gdzie założyliśmy statystyczną niezależność obserwacji. Znajdujemy więc:

$$P(O|\mathbf{q},\lambda) = b_{q_0}(o_0)b_{q_1}(o_1)...b_{q_{T-1}}(o_{T-1})$$

Prawdopodobieństwo sekwencji stanu $\mathbf{q}$ może być zapisane jako:https://www.overleaf.com/project/5f37f5ee9ecc1b000115

$$P(\mathbf{q}|\lambda) = \pi_{q_0}a_{q_0q_1}a_{q_1q_2}...a_{q_{T-2}q_{T-1}}$$

Prawdopodobieństwo, że O i $\mathbf{q}$ zachodzą jednocześnie, to iloczyn dwóch powyższych wyrażeń:

$$P(O,\mathbf{q}|\lambda) = P(O|\mathbf{q},\lambda)P(\mathbf{q}|\lambda)$$

Prawdopodobieństwo wystąpienia O (mając dany model) uzyskujemye sumując powyższe wyrażenie przez wszystkie możliwe sekwencje stanu $\mathbf{q}$. Mamy więc:

$$P(O|\lambda) = \sum_{\mathbf{q}} P(O|\mathbf{q},\lambda)P(\mathbf{q}|\lambda)$$

Po rozwinięciu otrzymujemy:

$$P(O|\lambda) = \sum_{q_0,q_1,...,q_{T-1}} \pi_{q_0}b_{q_0}(o_0)a_{q_0q_1}b_{q_1}(o_1)...a_{q_{T-2}q_{T-1}}b_{q_{T-1}}(o_{T-1})$$

2.3.1 Problem

Sumowanie wyrażenia powyżej zawiera $2T-1$ multiplikacji a istnieje N^T różnorodnych możliwych sekwencji stanu. Bezpośrednia metoda policzenia $P(O|\lambda)$ zawierałaby więc rząd $2TN^T$ multiplikacji. Nawet dla małych liczb, np. $N = 5$ i $T = 100$ oznaczałoby to w przybliżeniu 10^{72} multiplikacji.

2.3.2 Inny algorytm

Istnieje bardziej optymalny algorytm nazywany Algorytmem Naprzód-Do tyłu (ang. Forward-Backward) zdolny do rozwiązania powyższego problemu.

Procedura naprzód wygląda następująco. Rozważmy zmienną naprzód $\alpha_t(i)$:

$$\alpha_t(i) := P(o_0o_1...o_t, q_t = i|\lambda)$$

gdzie $i = 0, 1, ..., N-1$. $\alpha_i(t)$ jest zatem prawdopodobieństwem częściowej obserwacji sekwencji aż do czasu t i stanu i w czasie t, mając dany model.Można pokazać, że $\alpha_t(i)$ może być policzone w sposób następujący:

2.3.3 Inicjalizacja

$$\alpha_0(i) = \pi_i b_i(o_0)$$

gdzie $i = 0, 1, ..., N-1$

2.3.4 Rekurencja

Dla $t = 0, 1, ..., T-2$ i $j = 0, 1, ..., N-1$ mamy:

$$\alpha_{t+1}(j) = \Big(\sum_{i=0}^{N-1} \alpha_t(i) a_{ij} \Big) b_j(o_{t+1})$$

gdzie $j = 0, 1, ..., N-1$ i $t = 0, 1, ..., T-2$

2.3.5 Prawdopodobieństwo

Mamy:

$$P(\mathbf{O}|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i)$$

Faza inicjalizacji zawiera N multiplikacji. W fazie rekurencji sumowanie zawiera N multiplikacji plus jedna na wyrażenie poza nawiasem $b_j(o_{t+1})$.Musi to być wykonane dla $j = 0$ do $N-1$ i $t = 0$ do $T-2$, co daje w sumie dla kroku drugiego (rekurencji) $(N-1)N(T+1)$ multiplikacji. Krok trzeci (prawdopodobieństwo) nie uwzględnia multiplikacji.
Daje nam to całkowitą sumę multiplikacji jako :

$$N + N(N+1)(T-1)$$

całość rzędu N^2T w porównaniu do metody bezpośredniej $2TN^T$.

2.3.6 Zmienna do-tyłu

W podobny sposób możemy zdefiniować zmienną do-tyłu $\beta_t(i)$ jako:

$$\beta_t(i) := P(o_{t+1} o_{t+2} ... o_{T-1}, q_t = i|\lambda)$$

gdzie $i = 0, 1, ..., N-1$. To oznacza, że $\beta_i(t)$ jest prawdopodobieństwem obserwacji sekwencji od $t+1$ do $T-1$ mając dany stan i w czasie t przy użyciu modelu λ. Można pokazać, że $\beta_t(i)$ może być policzone w sposób następujący:

$$\beta_{T-1}(i) = 1$$

gdzie $i = 0, 1, ..., N-1$.Dla $t = T-2, T-1, ..., 1, 0$ i $i = 0, 1, ..., N-1$ mamy:

$$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)$$

$$P(O|\lambda) = \sum_{i=0}^{N-1} \pi_i b_i(o_0) \beta_0(i)$$

Obie metody mogą być użyte do obliczenia $P(O|\lambda)$ gdyż mają ten sam rząd złożoności przy wykonaniu tj. N^2T

2.4 Algorytm Viterbi

2.4.1 Zagadnienie

Algorytm Viterbi'ego służy do policzenia optymalnej (najbardziej prawdopodobnej) sekwencji stanu $(q_0, q_1, ..., q_{T-1})$ w ukrytym modelu Markova mając daną sekwencję obserwowanych wyników. Precyzyjniej, musimy znaleźć sekwencję stanu taką że prawdopodobieństwo wystąpienia sekwencji obserwacji:

$$(o_0, o_1, ..., o_{T-1})$$

z tej sekwencji stanu jest większe niż z każdej innej sekwencji stanu. Nasz problem zatem sprowadza się do znalezienia $\mathbf{q}$ które zmaksymalizuje $P(O, \mathbf{q}|\lambda)$.

2.4.2 Reformulacja problemu

Użytecznym będzie reformulacja zagadnienia w celu pokazania idei algorytmu Viterbi'ego. Rozważmy wyrażenie na $P(O, \mathbf{q}|\lambda)$

$$P(O, \mathbf{q}|\lambda) = P(O|\mathbf{q}, \lambda)P(\mathbf{q}|\lambda) = \pi_{q_0} b_{q_0}(o_0) a_{q_0 q_1} b_{q_1}(o_1) ... a_{q_{T-2} q_{T-1}} b_{q_{T-1}}(o_{T-1})$$

Definiujemy:

$$U(q_0, q_1, ..., q_{T-1}) := -\left(\ln(\pi_{q_0} b_{q_0}(o_0)) + \sum_{t=1}^{T-1} \ln(a_{q_{t-1} q_t} b_{q_t}(o_t)) \right)$$

Z powyższego wynika:

$$P(O, \mathbf{q}|\lambda) = \exp(-U(q_0, q_1, ..., q_{T-1}))$$

W wyniku tego, zagadnienie estymacji optymalnego stanu, tzn.

$$max_{q_t} P(O, q_0, q_1, ..., q_{T-1}|\lambda)$$

staje się identyczne z:

$$min_{q_t} U(q_0, q_1, ..., q_{T-1})$$

Dzięki tej reformulacji problemu możemy patrzeć na wyrażenia typu:

$$-\ln(a_{q_i q_j} b_{q_j}(o_t))$$

jako koszt związany z przejściem ze stanu q_i do stanu q_j w czasie t.

2.4.3 Inne spojrzenie na algorytm

Algorytm Viterbi'ego może zostać teraz opisany w sposób następujący: Załóżmy, że jesteśmy w stanie i i chcielibyśmy przejść do stanu j. Mówimy, że waga ścieżki ze stanu i do stanu j wynosi:

$$-\ln(a_{ij} b_j(o_t))$$

to znaczy: ujemny logarytm prawdopodobieństwa przejścia ze stanu i do stanu j i wybór symbolu obserwowanego o_t w stanie j. Tutaj o_t jest obserwowanym symbolem wybranym po "odzwiedzeniu" stanu j. To ten sam symbol, który pojawia się w sekwencji obserwacji:

$$O = (o_0, o_1, ..., o_{T-1})$$

Kiedy stan początkowy jest wybrany jako stan i to odpowiednia waga wynosi:

$$-\ln(\pi_i b_i(o_0))$$

Powyższe wyrażenie nazywamy wagą początkową. Definiujemy wagę sekwencji stanów jako sumę wag sąsiadujących stanów. Odpowiada to wymnożeniu odpowiadających sobie prawdopodobieństw. W tym momencie znalezienie optymalnej sekwencji jest kwestią znalezienia ścieżki (sekwencji stanów) minimalnej wagi dla której dana obserwacja zachodzi.

2.5 Odległości między HMM

Jeżeli chcemy porównać dwa ukryte modele Markova to potrzebna będzie miara na dystans pomiędzy nimi. Taka miara oparta jest na odległości Kullblack'a-Leibler'a pomiędzy dwoma funkcjami rozkładu prawdopodobieństwa.

Miara odległości Kullblacka'a-Leibler'a przedstawia się następująco: niech $\rho_1(x)$ oraz $\rho_2(x)$ będą dwoma funkcjami rozkładu prawdopodobieństwa (albo funkcjami masy prawdopodobieństwa). Wtedy miara Kullblack'a-Leibler'a może być użyta żeby znaleźć jak blisko siebie są dwa rozkłady prawdopodobieństwa.

Def. Miara odległości Kullblacka'a-Leibler'a $I(\rho_1, \rho_2)$ umożliwiająca poznanie jak blisko funkcja rozkładu prawdopodobieństwa $\rho_2(x)$ jest przy $\rho_1(x)$ w stosunku do $\rho_1(x)$ definiowana jest jako:

$$I(\rho_1, \rho_2) := \int_{-\infty}^{\infty} \rho_1(x) \ln\left(\frac{\rho_1(x)}{\rho_2(x)}\right) dx$$

Gdy $\rho_1(x)$ i $\rho_2(x)$ są funkcjami masy prawdopodobieństwa wtedy:

$$I(\rho_1, \rho_2) := \sum_x \rho_1(x) \ln\left(\frac{\rho_1(x)}{\rho_2(x)}\right)$$

Miara odległości Kullblacka'a-Leibler'a nie jest w ogólności symetryczna:

$$I(\rho_1, \rho_2) \neq I(\rho_2, \rho_1)$$

Jeżeli chcielibyśmy po prostu porównać ρ_1 i ρ_2 możemy zdefiniować symetryczną miarę odległości jako:

$$I_s(\rho_1, \rho_2) := \frac{1}{2}(I(\rho_1, \rho_2) + I(\rho_2, \rho_1))$$

Poprzez użycie miary odległości Kullblacka'a-Leibler'a dochodzimy do definicji miary odległości pomiędzy dwoma HMM. Dla HMM, funkcja rozkładu prawdopodobieństwa jest bardzzo skomplikowana, praktycznie można ją uzyskać tylko poprzez użycie rekurencyjnej procedury - algorytm forward-backward lub upward-downward. Nie ma zwięzłej formuły na odległość K-L dla tych modeli. Często, metoda Monte-Carlo jest wykorzystywana do wyznaczania całki powyżej.

3 Implementacja

3.1 Implementacja błądzenia losowego w Pythonie

random_walk

August 28, 2020

Typowym przykładem łańcuchu Markova jest "błądzenie losowe". Dany jest zbiór (przestrzeń stanowa):

[1]:
```
state_space = {0, 1, 2, 3, 4, 5}
```

gdzie 0 to początek układu, a 5 to koniec układu. Załóżmy, że pewna kobieta jest w dowolnym punkcie tej przestrzenii. Wówczas wykonuje krok w prawo z prawdopodobieństwem p albo na lewo z prawdopodobieństwem $q = 1 - p.$, chyba że jest w początku układu gdzie wykonuje krok w prawo do 1 albo gdy jest w punkcie 5 gdzie wykonuje krok w lewo do 4. Niech σ_t oznacza jej pozycję po t krokach. To jest właśnie przykład łańcuchu Markova z daną wyżej przestrzenią stanu.2 oznacza, że kobieta znajduje się w punkcie 2. Macierz przejścia A wynosi:

[2]:
```
import numpy as np

q = "q"
p = "p"

A = np.array([[0, 1, 0, 0, 0, 0], [q, 0, p, 0, 0, 0], [0, q, 0, p, 0, 0], [0,␣
 ↪0, q, 0, p, 0], [0, 0, 0, q, 0, p], [0, 0, 0, 0, 1, 0]])

print(A)
```

```
[['0' '1' '0' '0' '0' '0']
 ['q' '0' 'p' '0' '0' '0']
 ['0' 'q' '0' 'p' '0' '0']
 ['0' '0' 'q' '0' 'p' '0']
 ['0' '0' '0' 'q' '0' 'p']
 ['0' '0' '0' '0' '1' '0']]
```

Teraz zadajemy pytanie: Jakie jest prawdopodobieństwo, określone przez $a_{ij}^{(t)}$, że system zmienia się ze stanu q_i do q_j w ciągu t kroków? Niech A będzie macierzą przejścia dla procesu łańcuchu Markova. Wtedy macierz dla kroku t jest równa potędze A^t tj.

$A^{(t)} = A^t $

$a_{ij}^{(t)}$ to elementy macierzy $A^{(t)}$

Załóżmy, że kobieta zaczyna w punkcie 2. Żeby znaleźć rozkład prawdopodobieństwa po 3 krokach przeprowadzamy następujące rozumowanie:

Skoro $p^{(0)} = (0, 0, 1, 0, 0, 0)$

[3]:
```
p = 0.5
q = 0.5

p_0 = np.array([0, 0, 1, 0, 0, 0])
A = np.array([[0, 1, 0, 0, 0, 0], [q, 0, p, 0, 0, 0], [0, q, 0, p, 0, 0], [0,␣
↪0, q, 0, p, 0], [0, 0, 0, q, 0, p], [0, 0, 0, 0, 1, 0]])

def prob_distrib(matrix, p_0, t):

  i = 0
  while i < t:
    next_distrib = p_0.dot(matrix)

    p_0 = next_distrib


    i += 1

  return print(next_distrib)

prob_distrib(A, p_0, 3)
```

```
[0.    0.5   0.    0.375 0.    0.125]
```

3.2 Implementacja HMM C++

```
#include <iostream>
#include <string>
#include <cmath>

using namespace std;

class Data{
private:
    double **transitions;
    double **emissions;
    double *pi_transitions;

public:
    Data(int , int);

    double get_transition(int i, int j)
    {return transitions[i][j];}

    double get_emission(int i, int j)
    {return emissions[i][j];}

    double get_pi_transition(int i)
    {return pi_transitions[i];}

    void set_transition(int i, int j, double v)
    {transitions[i][j] = v;}

    void set_emission(int i, int j, double v)
    {emissions[i][j] = v;}

    void set_pi_transition(int i, double v)
    {pi_transitions[i] = v;}

};


Data::Data(int n=0, int m=0)
{
    transitions = new double*[n+1];
    for(int i=0; i<n+1; i++){
        transitions[i] = new double[n+1];
    }

    emissions = new double*[n+1];
    for(int i=0; i<n+1; i++){
        emissions[i] = new double[m + 1];
    }
    pi_transitions = new double[n+1];
}


class HMM{
private:
    int N, M, T;
    string o;
```

```
    double ** alpha_table;
    double** beta_table;
    double* alpha_beta_table;
    double* xi_divisor;
    Data* current;
    Data* reestimated;

public:
    HMM(int n, int m);

    void error(const string s)
    {cerr << "error: " << s << '\n';}

    void init(int s1, int s2, double value)
    {current -> set_transition(s1, s2, value);}

    void pi_init(int s, double value)
    {current -> set_pi_transition(s, value);}

    void o_init(int s, const char c, double value)
    {current -> set_emission(s, index(c), value);}

    double a(int s1, int s2)
    {return current -> get_transition(s1, s2);}

    double b(int state, int pos)
    {return current -> get_emission(state, index(o[pos - 1]));}

    double b(int state, int pos, string o)
    {return current -> get_emission(state, index(o[pos - 1]));}

    double pi(int state)
    {return current -> get_pi_transition(state);}

    double alpha(const string s);
    double beta(const string s);
    double gamma(int t, int i);
    int index(const char c);
    double viterbi(const string s, int *best_sequence);
    double** construct_alpha_table();
    double** construct_beta_table();
    double* construct_alpha_beta_table();
    double xi(int t, int i, int j);

    void reestimate_pi();
    void reestimate_a();
    void reestimate_b();
    double* construct_xi_divisor();
    void maximize(string training, string test);
    void forward_backward(string s);



};

HMM::HMM(int n=0, int m=0)
{
    N = n; M = n;
```

```
    current = new Data(n, m);
    reestimated = new Data(n, m);
}

double HMM::alpha(const string s)
{
    string out;
    double P = 0.0;
    out = s;
    int T1 = out.length();
    double* previous_alpha = new double[N+1];
    double* current_alpha = new double[N+1];

    //Initalization

    for(int i = 1; i<=N; i++)
    {previous_alpha[i] = pi(i)*b(i, 1, out);}

    //Induction

    for(int t=1; t<T1; t++){
        for(int j=1; j<=N; j++){
            double sum = 0.0;
            for(int i=1; i<=N; i++){
                sum += previous_alpha[i] *a(i, j);
            }
            current_alpha[j] = sum * b(j, t+1, out);
        }

        for(int c=1; c<=N; c++){
            previous_alpha[c] = current_alpha[c];
        }
    }

    // Termination

    for(int i = 1; i<=N; i++ ){
        P += previous_alpha[i];
    }

    return P;
}


double HMM::beta(const string s)
{
    double P = 0.0;
    o = s;
    int T = o.length();
    double* next_beta = new double[N + 1];
    double* current_beta = new double[N + 1];

    //Initalization

    for(int i=1; i<=N; i++){
        next_beta[i] = 1.0;
    }
```

```
    //Induction

    double sum;
    for(int t=T-1; t>=1; t-- ){
        for(int i=1; i<=N; i++){
            sum = 0.0;
            for(int j=1; j<=N; j++){
                sum += a(i, j)*b(j, t + 1) * next_beta[j];
            }
            current_beta[i] = sum;
        }

        for(int c=1; c<=N; c++){
            next_beta[c] = current_beta[c];
        }

    }

    // Termination

    for(int i = 1; i<=N; i++){
        P += next_beta[i]*pi(i)*b(i, 1);
    }
    return P;
}




double HMM::gamma(int t, int i)
{
    return (alpha_table[t][i]*beta_table[t][i])/(alpha_beta_table[t]);
}


int HMM::index(const char c){
    switch (c) {
    case 'H': return 0;
    case 'T': return 1;
    default: error("no_legal_input_symbol!");
    return 0;
    }
}


double HMM::viterbi(const string s, int best_path[])
{
    double P_star = 0.0;
    string o = s;
    int *help = new int;
    int T = o.length();
    double* previous_delta = new double[N + 1];
    double* current_delta = new double[N + 1];
    int** psi = new int*[T+1];
    for(int i=0; i<=T; i++) psi[i] = new int[N+1];

    // Initialization:
```

```
    for(int i=1; i<=N; i++)
    {previous_delta[i] = pi(i)*b(i, 1); psi[1][i] = 0;}

    double tmp, max;

    // Recursion

    for(int t=2; t<=T; t++){
        for(int j=1; j<=N; j++){
            max = 0.0;
            for(int i=1; i<=N; i++){
                tmp= previous_delta[i]*a(i, j);
                if(tmp >= max) {max = tmp; psi[t][j] = i;}
            }
            current_delta[j] = max*b(j, t);
        }
        for(int c=1; c<=N; c++){
            previous_delta[c] = current_delta[c];
        }
    }

    // Termination:

    for(int i =1; i<=N; i++){
        if(previous_delta[i] >= P_star){
            P_star = previous_delta[i];
            best_path[T] = i;
        }
    }

    // Extract best sequence

    for(int t=T-1; t>=1; t--){
        best_path[t] = psi[t+1][best_path[t+1]];
    }

    best_path[T+1] = ~1;
    return P_star;

}


double** HMM::construct_alpha_table()
{
    double** alpha_table = new double*[T+1];
    for(int i=0; i<=T+1; i++) alpha_table[i] = new double[N+1];

    // Initalization

    for(int i =1 ; i<=N; i++) alpha_table[1][i] = pi(i)*b(i, 1);


    // Induction

    for(int t=1; t<T; t++){
        for(int j=1; j<=N; j++){
            double sum = 0.0;
```

```
            for(int i=1; i<=N; i++){
                sum += alpha_table[t][i]*a(i, j);
            }
            alpha_table[t+1][j] = sum*b(j, t+1);
        }
    }

    return alpha_table;
}


double** HMM::construct_beta_table(){

    double **beta_table = new double*[T+1];
    for(int i=0; i<=T+1; i++){
        beta_table[i] = new double[N+1];
    }

    // Initalization

    for(int i=1; i<=N; i++){
        beta_table[T][i] = 1.0;

    }

    double sum;

    //Induction

    for(int t=T-1; t>=1; t--){
        for(int i=1; i<=N; i++){
            sum = 0.0;
            for(int j=1; j<=N; j++){
                sum += a(i, j)*b(j, t+1) * beta_table[t+1][j];

            }
            beta_table[t][i] = sum;
        }
    }

    //Termination

    for(int i=1; i<=N; i++){
        beta_table[1][i] = beta_table[1][i] * pi(i) * b(i, 1);
    }

    return beta_table;
}


double* HMM::construct_alpha_beta_table()
{
    double* alpha_beta_table = new double[T+1];


    for(int t=1;t<=T;t++){
        alpha_beta_table[t] = 0;
        for(int i=1; i<=N; i++){
```

```
            alpha_beta_table[t] += (alpha_table[t][i] * beta_table[t][i]);


        }
    }
    return alpha_beta_table;
}

double* HMM::construct_xi_divisor()
{
    xi_divisor = new double[T+1];
    double sum_j;

    for(int t=1; t<T; t++){
        xi_divisor[t] = 0.0;
        for(int i = 1; i<=N; i++){
            sum_j = 0.0;
            for(int j=1; j<=N; j++){
            sum_j += (alpha_table[t][i] * a(i, j) * b(j, t+1)
                    * beta_table[t+1][j]);
            }
            xi_divisor[t] += sum_j;
        }
    }

    return xi_divisor;
}


double HMM::xi(int t, int i, int j){
    return ((alpha_table[t][i]*a(i, j)*b(j, t+1)*beta_table[t+1][j])/
            (xi_divisor[t]));
}

void HMM::reestimate_pi()
{
    for(int i=1; i<=N; i++){
        reestimated->set_pi_transition(i, gamma(1, i));
    }
}

void HMM::reestimate_a()
{
    double sum_xi, sum_gamma;

    for(int i=1; i<=N; i++){
        for(int j=1; j<=N; j++){
            sum_xi = 0.0; sum_gamma = 0.0;
            for(int t=1; t<T; t++){sum_xi += xi(t, i, j);}
            for(int t=1; t<T; t++){sum_gamma += gamma(t, i);}
            reestimated->set_transition(i, j, (sum_xi / sum_gamma));
        }
    }
}



void HMM::reestimate_b()
```

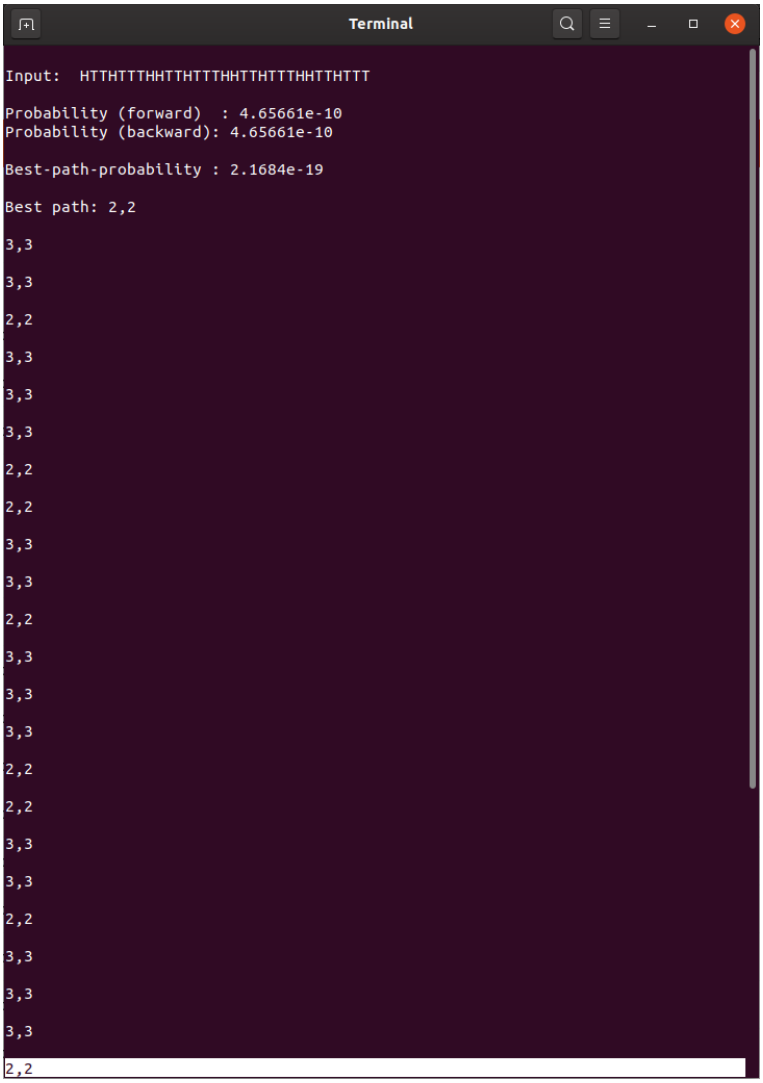
```
{
    double sum_gamma;
    double tmp_gamma;
    double sum_gamma_output;
    for(int j=1; j<=N; j++){
        for(int k=0; k<M; k++){
            sum_gamma = 0.0; sum_gamma_output = 0.0;

            for(int t=1; t<=T; t++){
                tmp_gamma = gamma(t, j);
                if(index(o[t-1]) == k){
                    sum_gamma_output += tmp_gamma;
                }
                sum_gamma += tmp_gamma;
            }

            reestimated->set_emission(j, k, (sum_gamma_output / sum_gamma));
        }
    }
}



void HMM::forward_backward(string o)
{
    T = o.length();
    alpha_table = construct_alpha_table();
    beta_table = construct_beta_table();
    alpha_beta_table = construct_alpha_beta_table();
    xi_divisor = construct_xi_divisor();
    reestimate_pi();
    reestimate_a();
    reestimate_b();

    //deletion

    for(int t=1; t<=T; t++){
        delete[] alpha_table[t];
    }

    delete[] alpha_table;

    for(int t=1; t<=T; t++){
        delete[] beta_table[t];
    }

        delete[] beta_table;
        delete[] alpha_beta_table;
        delete[] xi_divisor;

        Data* tmp_value = current;
        current = reestimated;
        reestimated = tmp_value;


}
```

```
void HMM::maximize(string o, string test)
{
    double diff_entropy, old_cross_entropy, new_cross_entropy;
    int c = 1;
    int t = test.length();
    old_cross_entropy = -((log10(alpha(test))/log10(2))/t);
    cout << "Reestimation:\n";
    cout << "␣initial␣cross_entropy:␣" << old_cross_entropy << "\n";

    do{
        forward_backward(o);
        new_cross_entropy = -(log10(alpha(test)) / log10(2)) / t;
        diff_entropy = (old_cross_entropy - new_cross_entropy);
        old_cross_entropy = new_cross_entropy;
        c++;
    }while(diff_entropy > 0.0);

    cout << "␣No␣of␣iterations:␣" << c << "\n";
    cout << "␣resulting␣cross_entropy:␣"
         << old_cross_entropy << "\n";
}



int main(void)
{

    HMM hmm(3, 2);
    hmm.pi_init(1, 0.33333333);
    hmm.pi_init(2, 0.33333333);
    hmm.pi_init(3, 0.33333333);

    hmm.init(1, 1, 0.33333333);
    hmm.init(1, 2, 0.33333333);
    hmm.init(1, 3, 0.33333333);
    hmm.init(2, 1, 0.33333333);
    hmm.init(2, 2, 0.33333333);
    hmm.init(2, 3, 0.33333333);
    hmm.init(3, 1, 0.33333333);
    hmm.init(3, 2, 0.33333333);
    hmm.init(3, 3, 0.33333333);

    hmm.o_init(1,'H',0.5);
    hmm.o_init(2,'H',0.75);
    hmm.o_init(3,'H',0.25);
    hmm.o_init(1,'T', 0.5);
    hmm.o_init(2,'T', 0.25);
    hmm.o_init(3,'T', 0.75);

    string training = "HTTHTTTHHTTHTTTHHTTHTTTHHTTHTTT";

    string test = "HTHTTHTHTTHTHTHTHTTHHTHTHTTHTHTTHHT";

    cout << "\nInput:␣␣" << training << "\n\n";
    cout << "Probability␣(forward)␣␣:␣"
         << hmm.alpha(training) << "\n";
    cout << "Probability␣(backward):␣"
```

```
        << hmm.beta(training) << "\n\n";
    int *best_path = new int[256];
    cout << "Best-path-probability : "
        << hmm.viterbi(training, best_path) << "\n\n";
    cout << "Best path: ";
    for(int t=1; best_path[t+1]!=-1; t++){
    cout << best_path[t] << "," ;
    cout << best_path[t] << "\n\n";}
    hmm.maximize(training, test);
    cout << " Probability (forward) : "
        << hmm.alpha(training) << "\n";
    cout << " Best-path-probability : "
        << hmm.viterbi(training, best_path) << "\n";
    return 0;

}
```

4 Wyniki i podsumowanie

Widzimy, że model ucząc sie przykładowego podanego wzorca i wykorzystując algorytmy określające prawdpodobieństwa przejścia spełnia swoje zadanie: znajduje prawdopodobieństwo obserwacji poprzez podanie mu modelu tj. string w inpucie programy oraz odnajduje najbardziej prawdopodobną sekwencję stanu poprzez podanie modelu oraz obserwacji.


Terminal
Input: HTTHTTTHHTTHTTTHHTTHTTTHHTTHTTT
Probability (forward) : 4.65661e-10
Probability (backward): 4.65661e-10
Best-path-probability : 2.1684e-19
Best path: 2,2
3,3
3,3
2,2
3,3
3,3
3,3
2,2
2,2
3,3
3,3
2,2
3,3
3,3
3,3
2,2
2,2
3,3
3,3
2,2
3,3
3,3
3,3
2,2

```
2,2

3,3

3,3

2,2

3,3

3,3

Reestimation:
 initial cross_entropy: 1
 No of iterations: 2
 resulting cross_entropy: 1.013
 Probability (forward) : 2.60327e-09
 Best-path-probability : 2.75955e-18
Press <RETURN> to close this window...
```