TARTU ÜLIKOOL

Matemaatika ja statistika instituut

Matemaatika ja statistika õppekava
Matemaatika eriala

Magistritöö (30 EAP)

Kolmekaupa Markovi ahelate Viterbi raja lähendamine

Oskar Soop

Juhendaja: Jüri Lember

TARTU 2023

Kolmekaupa Markovi ahelate Viterbi raja lähendamine

Magistritöö

Oskar Soop

Lühikokkuvõte.

Kolmekaupa Markovi mudelid (TMM) üldistavad varjatuid Markovi mudeleid (HMM) ja paarikaupa Markovi mudeleid (HMM). Nende mudelite puhul on tüüpiliseks ülesandeks vaatluste $y = (y_1, ..., y_n)$ põhjal varjatud protsessi $x = (x_1, ..., x_n)$ hindamine. Suurimat tõepäraga jada $\arg\max_x p(x|y)$ nimetatakse Viterbi rajaks ning varjatud ja paarikaupa Markovi mudelites on see leitav Viterbi algoritmiga. Erinevalt eelmainitud mudelitest, ei saa üldiselt kolmekaupa Markovi mudelitel kasutada Viterbi algoritmi ega selle üldistusi. Käesolevas magistritöös pakume välja mõned lähendusalgoritmid Viterbi raja lähendamiseks ning viime läbi numbrilised simulatsioonid, et võrrelda lähendusalgoritmide sooritusvõimet. Simulatsioonide tulemused näitavad, et praktikas on kõige mõistlikum kasutada k-Viterbi algoritmi.

CERCS teaduseriala: P160 Statistika, operatsioonianalüüs, programmeerimine, finants- ja kindlustusmatemaatika.

Märksõnad. HMM, PMM, TMM, juhuslikud protsessid, Markovi ahelad, Viterbi algoritm.

Approximating Viterbi path in triplet Markov Chains

Master's thesis

Oskar Soop

Abstract.

Triplet Markov models (TMM) generalize hidden Markov models (HMM) and pairwise Markov models (HMM). In these models, a typical task is to estimate the hidden process $x = (x_1, ..., x_n)$ based on observations $y = (y_1, ..., y_n)$. The most probable sequence $\arg\max_x p(x|y)$ is called the Viterbi path, and in hidden and pairwise Markov models, it can be found using the Viterbi algorithm. Unlike the two aforementioned models, the triplet Markov models generally cannot apply the Viterbi algorithm or its generalizations. In this master's thesis, we propose some approximation algorithms for approximating the Viterbi path and conduct numerical simulations to compare the performance of these algorithms. The simulation results indicate that in practice, it is most reasonable to use the k-Viterbi algorithm.

CERCS research specialisation: P160 Statistics, operations research, programming, actuarial mathematics.

Keywords. HMM, PMM, TMM, stochastic processes, Markov chains, Viterbi algorithm.

Sisukord

Sissejuhatus

Varjatud Markovi mudelid (HMM) leiavad laialt kasutust erinevates valdkondades: finantsmudelites, kõne tuvastamises, robootikas, juhtimissüsteemides, bioinformaatikas ja mujal. Varjatud Markovi mudel koosneb varjatud ehk meile teadmata realisatsiooniga juhuslikust protsessist $X = (X_1, ..., X_n) = X_1^n$ ja vaatlustest, meile teada oleva realisatsiooniga juhuslikust protsessist, $Y = Y_1^n$. Paar (X, Y) on HMM kui X on Markovi ahel ja iga $t = 1, ..., n$ korral $P(Y_t \in A | X_1 = x_1, ..., X_n = x_n) = P(Y_t \in A | X_1 = x_1)$. Tüüpiline ülesanne varjatud Markovi mudelitel on kõige tõenäolisema varjatud ahela X realisatsiooni leidmine – täpsemalt $\arg\max_{x_1^n} p(x_1^n | y_1^n)$. Viimast saab leida Viterbi algoritmiga ning me nimetame seda Viterbi rajaks.

Paarikaupa Markovi mudelid (PMM) üldistavad HMM-i, eeldades ainult, et (X, Y) on Markovi ahel. PMM puhul ei pruugi X ega Y olla Markovi ahelad ja seega on PMM-d võimelised kirjeldama keerulisemaid mudeleid kui HMM-d. Ka paarikaupa Markovi ahela puhul saab kasutada Viterbi algoritmi, et leida Viterbi rada.

Kolmekaupa Markovi mudelid (TMM) üldistavad PMM-i lisades mudelisse veel ühe varjatud stohhastilise protsessi $U = U_1^n$ nii, et (U, X, Y) on Markovi ahel, aga marginaalprotsessid U, X, Y, (U, X), (U, Y) ja (X, Y) ei pruugi olla Markovi ahelad [GGMP18]. Kui vaadata kolmekaupa Markovi mudelit protsessina (W, Y), kus $W = W_1^n = (U, X)$, taandub see paarikaupa Markovi mudelile. TMM üldistusvõime tuleneb sellest, et meid huvitav protsess (X, Y) ei pea olema Markovi ahel. Erinevalt HMM-st ja PMM-st kolmekaupa Markovi mudelite korral ei pruugi saada kasutada Viterbi algoritmi või selle üldistusi, et leida Viterbi rada $\arg\max_{x_1^n} p(x_1^n | y_1^n)$. Seega tuleb kasutada lähendusalgoritme.

Käesoleva magistritöö esimeses peatükis antakse ülevaade eri tüüpi Markovi ahelate definitsioonidest, omadustest ja näidetest. Teises peatükis tutvustatakse lähendusalgoritme k-Viterbi, k-blokk ning iteratiivseid lähendusalgoritme – segmenteerimis EM-algoritmi ja mäkke ronimise algoritmi. Idee kasutada segmenteerimis EM-algoritmi kolmekaupa Markovi ahelate puhul käidi välja artiklis [LGKK19, lk 160]. Lisaks tutvustatakse algoritmides kasutatavat dünaamilise planeerimise võtet ja edasi-tagasi algoritmi ning kuidas väikeste väärtustega arvutada. Kolmandas peatükis viime paaril kolmekaupa Markovi mudelil läbi numbrilised simulatsioonid, et võrrelda lähendusalgoritmide sooritusvõimet ning uurida nende omadusi.

1 Markovi ahelate definitsioon ja omadused

Selles peatükis defineerime Markovi ahelad, kus olekute hulgad ei pea olema ülimalt loenduvad. Veel defineerime paarikaupa ja kolmekaupa Markovi ahelad ning vaatame mõnda nende omadust. Alustame Markovi ahela üleminekumaatriksit üldistava konseptsiooniga, Markovi tuumaga, mis sisuliselt kujutab endast tingliku tõenäosust $P(Y \in B|X = x)$.

Definitsioon 1.1. Olgu $(\mathcal{X}, \mathcal{A})$ ja $(\mathcal{Y}, \mathcal{B})$ mõõtuvad ruumid. **Markovi tuumaks** (ka **üleminekutuumaks**) ruumist $(\mathcal{X}, \mathcal{A})$ ruumi $(\mathcal{Y}, \mathcal{B})$ nimetatakse funktsiooni $\kappa : \mathcal{X} \times \mathcal{B} \to [0, 1]$, mis rahuldab järgmisi omadusi:

1. iga fikseeritud $B \in \mathcal{B}$ korral on kujutus $x \mapsto \kappa(x, B)$ $\mathcal{A}$-mõõtuv funktsioon;
2. iga fikseeritud $x \in \mathcal{X}$ korral on $B \mapsto \kappa(x, B)$ tõenäosusmõõt ruumis $(\mathcal{Y}, \mathcal{B})$.

Olgu μ mõõt mõõtuvas ruumis $(\mathcal{Y}, \mathcal{B})$. Kui iga $x \in \mathcal{X}$ korral $\kappa(x, \cdot) \ll \mu$, siis Radon-Nikodymi teoreemi põhjal leidub tihedusfunktsioon $p(\cdot|x)$. Seega

$$\kappa(x, B) = \int_B p(y|x)\mu(dy).$$

Kui see tihedusfunktsioon on $\mathcal{A} \times \mathcal{B}$-mõõtuv, siis nimetame seda **üleminekutiheduseks**.

Näide 1 (Juhusliku ekslemise samm [Wik23e])**.** Olgu $\mathcal{X} = \mathcal{Y} = \mathbb{Z}$. Juhuslik ekslemine on protsess, kus olekust $x \in \mathbb{Z}$ liigutakse olekusse $x + 1$ tõenäosusega p ja $x - 1$ tõenäosusega $1 - p$. Sellele vastav tuum on

$$\kappa(x, B) = \sum_{b \in B} \kappa(x, \{b\}) = p\mathbf{1}_{\{x+1\}}(b) + (1 - p)\mathbf{1}_{\{x-1\}}(b),$$

kus $\mathbf{1}$ on indikaatorfunktsioon.

Näide 2 (Galton-Watsoni samm [Wik23e])**.** Olgu Markovi tuum κ mõõtuvast ruumist $(\mathbb{N}, \mathcal{P}(\mathbb{N}))$ ruumi $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$, kus $\mathcal{B}(\mathbb{R})$ on tavaline Borel σ-algebra reaalarvude hulgal. Siis

$$\kappa(n, B) = \begin{cases} \mathbf{1}_B(0), \text{ kui } n = 0, \\ P(\xi_1 + \xi_2 + ... + \xi_n \in B), \text{ kui } n > 0, \end{cases}$$

kus $\xi_1, ..., \xi_n$ on iid juhuslikud suurused.

Definitsioon 1.2. Juhuslikku protsessi $X = (X_1, X_2, ...)$ nimetatakse **Markovi ahelaks** üleminekutuumadega κ_i ja hulgal $\mathcal{X}_1$ antud algtõenäosusmõõduga π, kui iga $n \in \mathbb{N}$ ning $A_1 \in \mathcal{A}_1, ..., A_n \in \mathcal{A}_n$ korral

$$P(X_1 \in A_1, X_2 \in A_2, ..., X_n \in A_n) =$$
$$= \int_{x_1 \in A_1} ... \int_{x_{n-1} \in A_{n-1}} \pi(dx_1)\kappa_1(x_1, dx_2)...\kappa_{n-2}(x_{n-2}, dx_{n-1})\kappa_{n-1}(x_{n-1}, A_n).$$

Markovi ahelat nimetatakse **homogeenseks** kui Markovi tuum ei sõltu indeksist i ehk iga i korral $\kappa_i = \kappa$.

Kui homogeense Markovi ahela algjaotusel π on tihedus $p(x_1)$ ja Markovi tuumal on üleminekutihedus mõõdu μ suhtes, siis saame kirjutada[Ava21, lk 11]

$$P(X_1 \in A_1, X_2 \in A_2, ...,X_{n-1} \in A_{n-1}) =$$
$$= \int_{A_1 \times ... \times A_n} p(x_1)p(x_2|x_1)...p(x_n|x_{n-1})\mu^n(d(x_1, ..., x_n)).$$

Samasuguse võrduse saame kirja panna ka mittehomogeensel juhul, kuid siis üleminekutihedus sõltub indeksist ning korrutismõõt üle mille integreerime võib olla korrutismõõt üle erinevate mõõtude.

Tähistame ühistihedust kui $p(x_1^n) = p(x_1, ..., x_n) := p(x_1)p(x_2|x_1)...p(x_n|x_{n-1})$.

Näeme, et tihedused rahuldavad **Markovi omadust**

$$p(x_t|x_1^{t-1}) = p(x_t|x_1, ..., x_{t-1}) = \frac{p(x_1, ..., x_t)}{p(x_1, ..., x_{t-1})} = p(x_n|x_{t-1}). \tag{1}$$

ja seega ka

$$P(X_t \in A|X_1 = x_1, ..., X_{t-1} = x_{t-1}) = P(X_t \in A|X_{t-1} = x_{t-1}) = \kappa(x_{t-1}, A).$$

Definitsioon 1.3. Olgu hulk $(\mathcal{X}, \mathcal{A}, \mu_X)$ ja $(\mathcal{Y}, \mathcal{B}, \mu_Y)$ mõõduga ruumid. Hulgal $\mathcal{Z} \subset \mathcal{X} \times \mathcal{Y}$ koos korrutis-σ-algebraga $\mathcal{B}(\mathcal{Z}) = \mathcal{A} \otimes \mathcal{B}$ ja korrutismõõduga $\mu := \mu_X \times \mu_Y$ defineeritud homogeenset Markovi ahelat nimetatakse **paarikaupa Markovi mudeliks**, lühidalt **PMM**.

PMM-i korral ei pruugi marginaalprotsessid (X_t) ja (Y_t) eraldivõetuna olla Markovi ahelad, kuid tinglikute protsessidena $(X_t|Y_1^n)_{t=1}^n$ ja $(Y_t|X_1^n)_{t=1}^n$ on nad, üldiselt mittehomogeensed, Markovi ahelad

$$p(x_{t+1}|x_1^t, y_1^n) = \frac{p(x_{t+1}, y_{t+1}^n|x_1^t, y_1^t)}{p(y_{t+1}^n|x_1^t, y_1^t)} = \frac{p(x_{t+1}, y_{t+1}^n|x_t, y_t)}{p(y_{t+1}^n|x_t, y_t)} = p(x_{t+1}|x_t, y_t^n)$$
$$= p(x_{t+1}|x_t, y_1^n), \qquad \text{eeldades, et } p(z_1^n) = p(x_1^n, y_1^n) > 0.$$

Vastupidine väide ei kehti ehk kui $(X_t|Y_1^n)_{t=1}^n$ ja $(Y_t|X_1^n)_{t=1}^n$ on Markovi ahelad, siis (X_t, Y_t) ei pea olema Markovi ahel. Näiteks kui (X_t) ei ole Markovi ahel ja $(Y_t) = (X_t)$, siis (X_t, Y_t) ei ole Markovi ahel, aga $(X_t|Y_1^n)_{t=1}^n$ ja $(Y_t|X_1^n)_{t=1}^n$ on Markovi ahelad.

Tähistame $z_t = (x_t, y_t)$. Kasutades tingliku tiheduse omadusi, saame üleminekutihedust esitada kujul

$$p(z_{t+1}|z_t) = p(x_{t+1}, y_{t+1}|x_t, y_t) = p(y_{t+1}|x_{t+1}, x_t, y_t)p(x_{t+1}|x_t, y_t). \tag{2}$$

Kuju (2) võimaldab PMM-i klassifitseerida vastavalt sõltuvustele ning konstrueerida PMM-i tinglike üleminekutiheduste abil [Ava21, lk 16].

Näide 3. Varjatud Markovi mudeli, HMM-i, üleminekutihedus esitub kujul

$$p(z_2|z_1) = p(y_2|x_2)p(x_2|x_1).$$

Konkreetsema näitena võtame $\mathcal{X} = \{0, 1\}$ ja $\mathcal{Y} = \mathbb{R}$. Olgu $p(x_{t+1} = 1|x_t = 1) = 0.8$ ja $p(x_{t+1} = 0|x_t = 0) = 0.5$ ning $Y_t \sim N(X_t, 1)$.

Näide 4. Olgu meil suvaline mittehomogeenne Markovi ahel (Y_t), mis tähendab, et leiduvad indeksid, mille korral Markovi tuumad on erinevad. Mittehomogeensest Markovi ahelast saame teha homogeense Markovi ahela, kui me vaatame ahelat paarikaupa Markovi ahelana (T_t, Y_t), kus $P(T_t = t) = 1$.

Näide 5. Olgu stohhastiline protsess $(X_t)_{t=1}$ selline, et iga $t \in \mathbb{N}$ ja x_1^{t-1} korral leidub tinglik tihedusfunktsioon $p(x_t|x_1^{t-1})$, ja et ühistihedus on esitatav järgmiselt

$$p(x_1, x_2, .., x_n) = p(x_1)p(x_2|x_1)p(x_3|x_2, x_1)...p(x_n|x_{n-1}, ..., x_1).$$

Kui defineerime $Y_1 = ()$ ja $Y_t = (X_1, ..., X_{t-1})$, siis saame

$$\begin{aligned} p(x_1, .., x_n) &= p(x_1, .., x_n, y_1, ..., y_n) \\ &= p(x_1, y_1)p(x_2, y_2|x_1, y_1)p(x_3, y_3|x_2, y_2)...p(x_n, y_n|x_{n-1}, y_{n-1}), \end{aligned}$$

kus iga $t = 1, ..., n$ korral on $p(x_t, y_t|x_{t-1}, y_{t-1})$ üleminekutihedused. Järelikult on (X_t, Y_t) paarikaupa Markovi ahel.

Definitsioon 1.4. Kolmekaupa Markovi mudel ehk **TMM** on nagu PMM, aga defineeritud kolme hulga korrutise abil. See tähendab, et kui $(\mathcal{U}, \sigma_U, \mu_U)$, $(\mathcal{X}, \mathcal{A}, \mu_X)$ ja $(\mathcal{Y}, \mathcal{B}, \mu_Y)$ on mõõduga ruum, siis $(\mathcal{Z}, \mathcal{B}(\mathcal{Z}), \mu)$ on **TMM**, kus $\mathcal{Z} \subset \mathcal{U} \times \mathcal{X} \times \mathcal{Y}$, $\mathcal{B}(\mathcal{Z}) := \sigma_U \otimes \mathcal{A} \otimes \mathcal{B}$ ning $\mu := \mu_U \times \mu_X \times \mu_Y$. Selles töös üldiselt eeldame, et $\mathcal{X}$ ja $\mathcal{U}$ on lõplikud, diskreetsete σ-algebragatega $2^{\mathcal{X}}$ ja $2^{\mathcal{U}}$ ning loendava mõõduga.

Tähistame nüüd ja edaspidi $z_t = (u_t, x_t, y_t)$.

Kasutades tinglike tihedusi, saame üleminekutihedust esitada kujul

$$\begin{aligned} p(z_{t+1}|z_t) &= p(u_{t+1}, x_{t+1}, y_{t+1}|u_t, x_t, y_t) = \\ &= p(y_{t+1}|u_{t+1}, x_{t+1}, u_t, x_t, y_t)p(x_{t+1}|u_{t+1}, u_t, x_t, y_t)p(u_{t+1}|u_t, x_t, y_t). \end{aligned} \quad (3)$$

Nii paarikaupa Markovi ahelad kui ka kolmekaupa Markovi mudelid on Markovi ahelad ning et kolmekaupa Markovi ahel on ka paarikaupa Markovi ahel kui näiteks võtta hulgaks $\mathcal{X}$ PMM-i definitsioonis hulk $\mathcal{U} \times \mathcal{X}$ TMM-i definitsioonis. Samas marginaalprotsessid (U_t), (X_t), (Y_t), (U_t, X_t), (U_t, Y_t) ja (X_t, Y_t) ei pruugi olla Markovi ahelad.

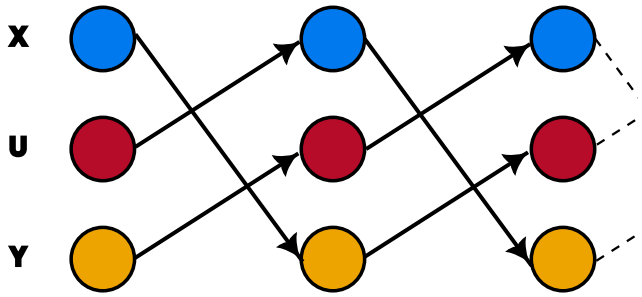
Näide 6. Heeliksi kujulise ahela korral (Vt. joonist), üleminekutihedusega

$$p(z_t|z_{t-1}) = p(u_t|y_{t-1})p(x_t|u_{t-1})p(y_t|x_{t-1}),$$

ei pruugi marginaalprotsessil kehtida Markovi omadus (1)

$$p(x_t|x_{t-1}, x_{t-2}, x_{t-3}) = p(x_t|x_{t-3}) \neq p(x_t|x_{t-1}) = p(x_t)$$

ja seega ei ole alati tegu Markovi ahelaga.



Joonis 1: Heeliks ahel. Ahel, mille marginaalprotsessid ei pruugi olla Markov

Sellegipoolest heeliksi kujulise ahela igal marginaalprotsessil on niinimetatud 3-astme Markovi omadus

$$p(x_t|x_1^{t-1}) = p(x_t|x_{t-1}, x_{t-2}, x_{t-3}),$$

mis võimaldab kasutada Markovi ahelatele sarnast teooriat, sealjuures protsess $(X_t, X_{t-1}, X_{t-2})_t$ on Markovi ahel.

2 Algoritmid

Kõiges järgnevas tähistame tähega p erinevaid tihedusi. Näiteks $p(z_t) = p(w_t, y_t) = p(u_t, x_t, y_t)$ tähistab $Z_t = (U_t, X_t, Y_t)$ tihedust väärtusel $z_t = (u_t, x_t, y_t)$ ning $p(z_1^n) = p(z)$ tähistab ühistihedust väärtustel $z = z_1^n = (z_1, ..., z_n)$.

Kui argumentidena kasutada z_t, w_t, u_t, x_t või y_t asemel mõnda teist sümbolit, siis kasutame selle esitamiseks võrdusmärki, näiteks $p(x_2 = i|u_2 = j)$.

Edaspidi eeldame, et kolmekaupa Markovi ahela $(Z_t)_{t=1}^n = (U_t, X_t, Y_t)_{t=1}^n$ olekute hulgad $\mathcal{U}$ ja $\mathcal{X}$ on lõplikud ehk $|\mathcal{U}| < \infty$ ja $|\mathcal{X}| < \infty$. Lisaks eeldame, et marginaalprotsessi $(Y_t)_{t=1}^n$ realisatsioon y_1^n on teada ning $p(y_1^n) > 0$.

Kui naiivselt leida Viterbi rada – x_1^n, mis maksimiseerib tihedust $p(x_1^n|y)$ – proovides läbi kõik võimalikud ahela realisatsioonid, peab selleks arvutama $|\mathcal{X}|^n$ tiheduse väärtust, mis on praktiline ainult väga väikeste ahelate pikkuste korral. Paarikaupa Markovi ahelate puhul saab kasutada Viterbi algoritmi (vt ptk 2.4), millega leiab Viterbi raja $O(n|\mathcal{X}|^2)$ elementaaroperatsiooniga[1]. Kuna kolmekaupa Markovi ahelas ei pruugi tinglik protsess $(X_t|Y_1^n)_{t=1}^n$ olla Markovi ahel, ei saa või ei oska me marginaalprotsessi efektiivselt faktoriseerida ja seega ei saa kasutada Viterbi algoritmi või selle üldistust (vt ptk 2.4 ja ptk 2.1 näide 8).

2.1 Dünaamiline planeerimine

Kui ülesandel, mida me algoritmiga lahendame, on optimaalseid alamstruktuure — mis tähendab, et ülesande optimaalse lahenduse saab koostada alamülesannete optimaalsetest lahendustest — ja kattuvaid alamülesandeid, mis tähendab, et samu alamülesandeid kasutatakse paljude erinevate ülesannete lahendamiseks, väldib kiirem lähenemine, mida nimetatakse **dünaamiliseks planeerimiseks**, lahenduste uuesti arvutamist, kui need on juba välja arvutatud[Vik23]. Dünaamiline planeerimine on algoritmide koostamise strateegia/paradigma ning seda on kõige parem illustreerida näidetega.

Näide 7. [TG17] Fibonacci jada F_n on defineeritud järgmiselt:

$$F_n = \begin{cases} 0, & \text{kui } n = 0, \\ 1, & \text{kui } n = 1, \\ F_{n-1} + F_{n-2} & \text{kui } n > 1. \end{cases}$$

[1] Elementaaroperatsioonideks loeme liitmist, korrutamist ja kahe väärtuse omavahelist võrdlemist. Kui $f(m)$ on elementaaroperatsioonide arv, siis $O(g(m))$ tähendab, et leidub m_0 ja M nii, et iga $m > m_0$ korral kehtib $|f(m)| \leq Mg(m)$.

Viga, mida arvatavasti keegi käsitsi arvutades ei tee, aga programmeerides võib juhtuda on see, et Fibonacci arvu F_n jaoks arvutatakse F_{n-1} ja F_{n-2} eraldi, kasutamata F_{n-1} arvutamiseks juba olemasolevat F_{n-2} arvutust.

```
def Fibonacci(n):
    if(n==0):
        return 0
    if(n==1):
        return 1
    return Fibonacci(n-1) + Fibonacci(n-2)
```

Algoritm 1: Python example

Palju kiirema algoritmi saab, kui hoida varasemaid arvutusi mälus.

```
A = [0]*n #konstrueerime pikkusega n massiivi,
          #mille elemendid on nullid

def Fibonacci(n):
    if(n==0):
        return 0
    if(n==1):
        return 1
    if(A[n-1] == 0):
        A[n-1] = Fibonacci(n-1) + Fibonacci(n-2)
    return A[n-1]
```

Algoritm 2: Python example

Siin varasemate arvutuste mälus hoidmist ja nende kasutamist võibki pidada dünaamilise planeerimisega probleemi lahendamiseks. Märgime ära, et tegelikult ei pea Fibonacci arvu F_n leidmiseks kõiki varasemaid arvutusi meelde jätma, vaid piisab kahest viimasest F_{n-1} ja F_{n-2}.

Näide 8. [Loe04]

Olgu $f : M^7 \to \mathbb{R}$, kus M on lõplik hulk suurusega m, tegurdatav järgmiselt:

$$f(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = f_1(x_1)f_2(x_2)f_3(x_1, x_2, x_3, x_4) \cdot f_4(x_4, x_5, x_6)f_5(x_5)f_6(x_6, x_7, x_8)f_7(x_7).$$

Ülesanne on leida

$$\max_{(x_1,x_2,x_3,x_4,x_5,x_6,x_7)} f(x_1,x_2,x_3,x_4,x_5,x_6,x_7).$$

Üks variant on proovida kõiki m^7 võimalikke $(x_1,x_2,x_3,x_4,x_5,x_6,x_7)$ olekuid ja vaadata, milline neist saavutab maksimumi.

Dünaamilise planeerimisega lahendamiseks, paneme tähele, et see ülesanne on sama kui

$$\max_{x_1}\max_{x_2}\max_{x_3}\max_{x_4}\max_{x_5}\max_{x_6}\max_{x_7} f(x_1,x_2,x_3,x_4,x_5,x_6,x_7).$$

Liigudates maksimumi võtmise korrutamise vahele saame näiteks, et

$$\max_{x_4}\Bigg(\Bigg[\max_{(x_1,x_2,x_3)} f_1(x_1)f_2(x_2)f_3(x_1,x_2,x_3,x_4)\Bigg] \cdot \Bigg[\max_{(x_5,x_6)} f_4(x_4,x_5,x_6)f_5(x_5)\Big(\max_{(x_7,x_8)} f_6(x_6,x_7,x_8)f_7(x_7)\Big)\Bigg]\Bigg)$$

Tähistame

$$\mu_3(x_6) = \max_{(x_7,x_8)} f_6(x_6,x_7,x_8)f_7(x_7),$$

$$\mu_2(x_4) = \max_{(x_5,x_6)} f_4(x_4,x_5,x_6)f_5(x_5)\mu_3(x_6),$$

$$\mu_1(x_4) = \max_{(x_1,x_2,x_3)} f_1(x_1)f_2(x_2)f_3(x_1,x_2,x_3,x_4),$$

$$\mu = \max_{x_4} \mu_1(x_4)\mu_2(x_4).$$

Näeme, et μ_3, μ_2, μ_1 ja μ leidmiseks tuleb proovida vastavalt läbi proovida m^3, m^3, m^4 ja m olekuid, mis kokku on $m^4 + 2m^3 + m = O(m^4)$ oleku läbivaatamist. See on suurusjärku m^3 korda vähem läbivaatamisi kui proovides kõiki m^7 võimalikke $(x_1,x_2,x_3,x_4,x_5,x_6,x_7)$ olekuid.

Kui maksimiseerimise asemel on summeerimine, siis saab täpselt sama moodi leida summat dünaamilise planeerimise abil.

2.2 Väikeste tihedustega arvutamine

Algoritmides kasutatavad tihedused on väga väikesed (nt. e^{-800}), mistõttu praktikas kasutatakse tiheduste asemel Log tihedusi.

Olgu $p_1, ..., p_n$ tihedused ja $q_1 = \ln p_1, ..., q_n = \ln p_n$ vastavad Log tihedused. Lepime kokku, et $\ln 0 = -\infty$, $e^{-\infty} = 0$ ja $0 \ln 0 = 0$. Log tihedusi kasutades muutub tihedus-

te korrutamine Log tiheduste liitmiseks: $\ln(p_1 p_2) = \ln(p_1) + \ln(p_2)$. Samas väikeste tiheduste summeerimine tavapärase valemiga

$$\ln \sum_{i=1}^{n} p_i = \ln \sum_{i=1}^{n} \exp q_i,$$

on ebatäpne. Seepärast kasutatakse tõenäosuste summeerimiseks **LogSumExp** trikki[Wik23d]. Olgu $q^* := \max\{q_1, ..., q_n\}$. Trikiks on summa leida järgmiselt:

$$\ln \sum_{i=1}^{n} p_i = q^* + \ln \sum_{i=1}^{n} \exp\,(q_i - q^*).$$

Triki idee on nihutada reaaltelge logaritmisel skaalal nii, et liidetavad oleksid logaritmisel skaalal nulli lähedal.

Selle töö raames teostatud algoritmid kasutavad Log tihedusi ning algoritmides tiheduste liitmiseks kasutatakse LogSumExp trikki. Näitame, kuidas kasutame Log tihedusi LogSumExp trikki edasi-tagasi algoritmi korral, mida tutvustatakse peatükis 2.3. Olgu $w_t = (u_t, x_t)$. Edasi-tagasi algoritm koosneb abifunktsioonidest

$$\alpha_t(w_t) = p(w_t, y_1^t) = \sum_{w_{t-1}} p(z_t|z_{t-1})\alpha_{t-1}(w_{t-1}))$$

ja

$$\beta_t(w_t) = p(y_{t+1}^n|z_t) = \sum_{w_{t+1}} \beta_{t+1}(w_{t+1})p(z_{t+1}|z_t),$$

mille abil leitakse $p(w_t, y) = \alpha_t(w_t)\beta_t(w_t)$.

Kuna abifunktsioonid α_t ja β_{n-t} üldiselt indeksi t kasvades kahanevad eksponentsiaalselt nulli, siis suurte indeksite t korral võib esineda arvutustes allakadu (ingl. *underflow*) [DP04]. Kasutades Log tihedusi ja LogSumExp trikki saame allakadust hoiduda, esitades edasi-tagasi algoritmi kujul

$$\ln \alpha_1(w_1) = \ln p(z_1),$$
$$\ln \beta_n = 0$$

ja

$$\ln \alpha_t(w_t) = \ln p(w_t, y_1^t) = \alpha_t^* + \ln \sum_{w_{t-1}} \exp(\ln \alpha_{t-1}(w_{t-1}) + \ln p(z_t|z_{t-1}) - \alpha_t^*),$$
$$\ln \beta_t(v_t) = \ln p(y_{t+1}^n|z_t) = \beta_{t+1}^* + \ln \sum_{w_{t+1}} \exp(\ln \beta_{t+1}(w_{t+1}) + \ln p(z_{t+1}|z_t) - \beta_{t+1}^*),$$

kus

$$\alpha_t^* = \max_{w_t, w_{t-1}} (\ln \alpha_{t-1}(w_{t-1}) + \ln p(z_t|z_{t-1}))$$

ja

$$\beta_{t+1}^* = \max_{w_t, w_{t-1}} (\ln \beta_{t+1}(w_{t+1}) + \ln p(z_{t+1}|z_t)).$$

Leitud abifunktsioone $\ln \alpha_t$ ja $\ln \beta_t$ abil saame leida Log tiheduse

$$\ln p(w_t, y) = \ln \alpha_t(w_t) + \ln \beta_t(w_t).$$

Veel on olemas teine meetod, millega saab väikeseid tiheduste väärtuseid mingil määral kontrolli all hoida, kasutades ühistiheduste asemel tinglike tihedusi. Sellisel eesmärgil tinglike tiheduste kasutamist nimetatakse normaliseerimiseks[DP04] või **skaleerimiseks**[KL22]. Näitena toodud edasi-tagasi algoritmi saab esitada skaleeritud kujul, nagu esitatud artiklis [DP04]:

$$\overline{\alpha}_1(w_1) = \frac{p(z_1)}{\sum_{w_1} p(z_1)},$$
$$\overline{\alpha}_t(w_t) = p(w_t|y_1^t) = \frac{\sum_{w_{t-1}} p(z_t|z_{t-1})\overline{\alpha}_{t-1}(w_{t-1})}{\sum_{w_{t-1}, w_t} p(z_t|z_{t-1})\overline{\alpha}_{t-1}(w_{t-1}))}$$

ja

$$\overline{\beta}_n = 1,$$
$$\overline{\beta}_t(w_t) = \frac{p(y_{t+1}^n|w_t, y_t)}{p(y_{t+1}^n|y_1^t)} = \frac{\sum_{w_{t+1}} \overline{\beta}_{t+1}(w_{t+1})p(z_{t+1}|z_t)}{\sum_{w_t, w_{t+1}} \alpha_t(w_t)p(z_{t+1}|z_t)}.$$

Abifunktsioonide $\overline{\alpha}_t$ ja $\overline{\beta}_t$ saame leida tiheduse

$$p(w_t|y) = \overline{\alpha}_t(w_t)\overline{\beta}_t(w_t).$$

Kuna skaleerimata kujul on algoritmid lihtsamini kirja pandavad ning Log tiheduste kasutamisest mõistlike ahela pikkuste puhkudel peaks piisama, siis selles töös skaleeritud kuju ei kasutata.

2.3 Edasi-tagasi algoritm

Edasi-tagasi algoritmi abil saab leida marginaaltõepärasid kujul $p(w_t, y)$, kus $w_t = (u_t, x_t)$ [LLLP11]. Toome selle algoritmi eraldi välja, sest see leiab rakendust igas järgnevas algoritmis. Näiteks selle algoritmiga saab leida tihedusi $p(y)$, $p(x, y)$, $p(x_t, y)$. Kirjeldame algoritmi juhul kui y on fikseeritud ning $w = (u, x)$ on muutuja, samas kui me tahame leida tihedust $p(x, y)$, siis fikseerime (x, y) ning u on muutja.

Algoritm koosneb edasi liikuvast osast

$$\alpha_t(w_t) := p(w_t, y_1^t) = \sum_{w_{t-1}} p(w_{t-1}^t, y_1^t) = \sum_{w_{t-1}} p(z_t|w_{t-1}, y_1^{t-1}) p(w_{t-1}, y_1^{t-1}) =$$

$$= \sum_{w_{t-1}} p(z_t|z_{t-1}) \alpha_{t-1}(w_{t-1}),$$

$$\alpha_1(w_1) = p(z_1)$$

ja tagasi liikuvast osast

$$\beta_t(w_t) := p(y_{t+1}^n|z_t) = \sum_{w_{t+1}} p(y_{t+2}^n|z_t^{t+1}) p(z_{t+1}|z_t) = \sum_{w_{t+1}} \beta_{t+1}(w_{t+1}) p(z_{t+1}|z_t),$$

$$\beta_n = 1.$$

Neid omavahel korrutades saame marginaaltõepära

$$p(w_t, y) = p(w_t, y_1^t) p(y_{t+1}^n|z_t) = \alpha_t(w_t)\beta_t(w_t)$$

ja siis summeerides üle w_t või u_t saame

$$p(y) = \sum_{w_t} p(w_t, y), \qquad p(x_t, y) = \sum_{u_t} p(w_t, y).$$

Samamoodi saame, et

$$p(x, y) = \sum_{u_t} p(u_t, x, y).$$

Definitsioon 2.1. [KL22, LK14] Kui me leiame ${}^{\text{PMAP}}x \in \mathcal{X}^n$ nii, et iga t korral ${}^{\text{PMAP}}x_t = \arg\max_{x_t} p(x_t, y)$, siis nimetatakse seda **PMAP** rajaks.

Nimetame PMAP raja leidmist 0-Viterbi algoritmiks, sest tegu on peatükis 2.4 käistletava k-Viterbi algoritmi ühe juhtumina. Märgime ära, et PMAP rada kattub keskmiselt kõige rohkem tingliku protsessi $(X_t|Y_1^n)_{t=1}^n$ realisatsiooniga. Täpsemalt

$$^{\text{PMAP}}x = \arg\max_{\hat{x}} \mathbf{E}\left[\sum_{t=1}^{n} \mathbf{I}_{\{\hat{x}_t\}}(X_t)\middle| Y_1^n = y_1^n\right],$$

kus I on indikaatorfunktsioon. PMAP rada ei pruugi olla positiivse tihedusega $p(^{\text{PMAP}}x, y)$ [LK14].

2.3.1 k-edasi-tagasi algoritm

Edasi-tagasi algoritmi saab iga $k \in \mathbb{N}$ korral üldistada marginaaltõepärade $p(w_t, w_{t+1}, ..., w_{t+k-1}, y)$ leidmiseks, kus $w_t = (u_t, x_t)$.

Edasi algoritmiks on kaks varianti.

1.Olgu $t > 1$. Kui $t = 0$, siis kasutab teist edasi algoritmi varianti.

$$\alpha_t^k(w_t^{t+k-1}) := p(w_t^{t+k-1}, y_1^{t+k-1}) = \sum_{w_{t-1}} p(w_{t-1}^{t+k-1}, y_1^{t+k-1}) =$$
$$= \sum_{w_{t-1}} p\left(z_{t+k-1}|z_{t+k-2}\right) p(w_{t-1}^{t+k-2}, y_1^{t+k}) = p\left(z_{t+k-1}|z_{t+k-2}\right) \sum_{w_{t-1}} \alpha_{t-1}^k(w_{t-1}^{t+k-2})$$

Hindame selle algoritmi ajalist keerukust, loendades liitmisi ja korrutamisi. Fikseeritud k korral on $n-k$ blokki v_t^{t+k-1}, kus igas blokis on $|\mathcal{U} \times \mathcal{X}|^k$ erinevat olekut. Summa koosneb $|\mathcal{U} \times \mathcal{X}|$ liidetavast. Saame, et keerukus on $O(n\,|\mathcal{U} \times \mathcal{X}|^{k+1})$.

2. Olgu $k > 1$. Kui $k = 1$, siis kasutab klassikalist edasi algoritmi.

$$\begin{aligned}\alpha_t^k(w_t^{t+k-1}) :=& p(w_t^{t+k-1}, y_1^{t+k-1}) = \\ =& p\left(w_{t+k-1}, y_{t+k-1}|w_t^{t+k-2}, y_1^{t+k-2}\right) p(w_t^{t+k-2}, y_1^{t+k-2}) = \\ =& p\left(z_{t+k-1}|z_{t+k-2}\right) \alpha_t^{k-1}(w_t^{t+k-2})\end{aligned}$$

Hindame selle algoritmi keerukust. Klassikalise edasi algoritmiga α^1 leidmine on keerukusega $O(n|\mathcal{U} \times \mathcal{X}|^2)$. Liikumine $\alpha_t^{k-1} \to \alpha_t^k$ võtab $O(|\mathcal{U} \times \mathcal{X}|^k)$ operatsiooni. Seega liikumine $\alpha_t^1 \to \alpha_t^k$ võtab $O(k\,|\mathcal{U} \times \mathcal{X}|^k)$ operatsiooni. Kokkuvõttes on selle algoritmi keerukus $O(nk\,|\mathcal{U} \times \mathcal{X}|^k)$.

Soovitame edasi algoritmi valida järgmiselt. Kui me tahame leida $\alpha^1, ..., \alpha^k$, siis on parem kasutada 2. varianti. Kui me tahame leida ainult α^k ja $k > |\mathcal{U} \times \mathcal{X}|$, siis on mõistlikum kasutada 1. varianti.

Tagasi algoritm taandub juhule $k = 1$.

$$\beta_t^k(w_t^{t+k-1}) := p\left(y_{t+k}^n | z_t^{t+k-1}\right) = p\left(y_{t+k}^n | z_{t+k-1}\right) = \beta_{t+k-1}^1(w_{t+k-1})$$

Bloki tõenäosuse leiame järgmiselt:

$$p(x_t^{t+k-1}, y) = \sum_{u_t^{t+k-1}} p(w_t^{t+k-1}, y) = \sum_{u_t^{t+k-1}} \alpha_t^k(w_t^{t+k-1}) \beta_{t+k-1}^1(w_{t+k-1}).$$

2.4 k-Viterbi algoritm

Eesmärgiks on leida rada x, mis maksimiseerib tihedust $p(x|y)$. Kasutades Bayesi valemit saame, et

$$p(x|y) = p(x_1|y)p(x_2|x_1, y)p(x_3|x_1, x_2, y) \cdots p(x_n|x_1, ..., x_{n-1}, y).$$

Analoogselt dünaamilise planeerimise näitele (ptk 2.1 näide 8) ei ole võimalik efektiivselt maksimumidega tegurite vahele liikude, sest leidub tegur, mis sõltub kõikidest n muutujast $(x_1, ..., x_n)$ üle mille me maksimiseerime. Kui teeme protsessi $(X_t|Y_1^n)_{t=1}^n$ osas eeldusi, siis on võimalik sõltuvusi kaotada ning efektiivselt kasutada dünaamilist planeerimist, et leida otsitav maksimum.

Näiteks, kui tinglik marginaalprotsess $(X_t|Y_1^n)_{t=1}^n$ on Markovi ahel, siis kehtib Markovi omadus (1) ning võime kirjutada

$$p(x|y) = p(x_1|y)p(x_2|x_1, y) \cdots p(x_n|x_{n-1}, y)$$

ning maksimiseerimülesannet saame kirjutada kui

$$\max_x p(x|y) = \max_{x_n} \left(... \max_{x_2} \left(\max_{x_1} \Big[p(x_1|y)p(x_2|x_1, y) \Big] p(x_3|x_2, y) \right) \cdots p(x_n|x_{n-1}, y) \right),$$

kus iga maksimumivõtmise jaoks (v.a muutuja x_n jaoks) tuleb läbi vaadata $|\mathcal{X}|^2$ olekut. Igal sammul saame maksimiseerimise kirja panna järgmiselt:

$$\delta_1(x_2) = \max_{x_1} p(x_1|y)p(x_2|x_1, y),$$

$$\delta_t(x_{t+1}) = \max_{x_t} \delta_{t-1}(x_t)p(x_{t+1}|x_t, y), \quad \text{kui } t = 2, ..., n-1,$$

$$\delta_n = \max_{x_n} \delta_{n-1}(x_n).$$

Kui lisaks igal sammul maksimiseerijad meelde jätta maksimiseerijad

$$\gamma_1(x_2) = \arg\max_{x_1} p(x_1|y)p(x_2|x_1, y),$$

$$\gamma_t(x_{t+1}) = \arg\max_{x_t} \gamma_{t-1}(x_t)p(x_{t+1}|x_t, y), \quad \text{kui } t = 2, ..., n-1,$$

$$\gamma_n = \arg\max_{x_n} \gamma_{n-1}(x_n),$$

siis on võimalik tagasiliikumise teel leida maksimiseeriva raja $\hat{x}_n$:

$$\hat{x}_n = \gamma_n$$

ja iga $t = n-1, ..., 1$ korral

$$\hat{x}_t = \gamma_t(x_{t+1}).$$

Niimoodi maksimiseeriva raja leidmist nimetatakse **Viterbi algoritmiks** [LGKK19].

Analoogselt kui on tegu k-astme Markovi ahelaga, see tähendab, et juhuslikul protsessil on k-astme Markovi omadus $p(x_t|x_1, x_2, ..., x_{t-1}, y) = p(x_t|x_{t-k}, ..., x_{t-1}, y)$, siis võime kirjutada

$$p(x|y) = p(x_1, ..., x_k|y)p(x_{k+1}|x_1, ..., x_k, y) \cdots p(x_n|x_{n-k}, ..., x_{n-1}, y).$$

Parempoolset korrutist saab maksimiseerida analoogselt Viterbi algoritmile ning nimetame selle **k-Viterbi algoritmiks**. Algoritmi skeem on esitatud peatükis 2.5.1.

Nii paarikaupa kui ka kolmekaupa Markovi ahelate puhul saab tinglikud tihedused $p(x_t|x_{t-k}, ..., x_{t-1}, y) = \frac{p(x_{t-k},...,x_{t-1},x_t,y)}{p(x_{t-k},...,x_{t-1},y)}$ leida k-edasi-tagasi algoritmiga, sealhulgas mainime ära, et praktilistel kaalutlustel viimases jagatises nulliga jagamine on null ehk $\frac{0}{0} = 0$, sest kui $p(x_{t-k}, ..., x_{t-1}, y) = 0$, siis ka $p(x_{t-k}, ..., x_{t-1}, x_t, y) = 0$ ja $p(x_1^n, y) = 0$.

Näeme, et Viterbi algoritm on sama kui 1-Viterbi algoritm ning nimetame edasi-tagasi algoritmiga leitavate marginaaltõenäosuste korrutise $p(x_1|y)p(x_2|y) \cdots p(x_n|y)$ maksimiseerimist **0-Viterbi algoritmiks**. Viimasena mainitud algoritm leiab PMAP raja (definitsioon 2.1).

Kui juhuslik protsess $(X_t, Y_t)_{t=1}^n$ on paarikaupa Markovi ahel, siis tinglik protsess $(X_t|Y_1^n)_{t=1}^n$ on Markovi ahel ja seega saab Viterbi algoritmi abil leida $\arg\max_x p(x|y)$.

Kolmekaupa Markovi ahela korral tinglik protsess $(X_t|Y_1^n)_{t=1}^n$ ei pruugi olla Markovi ahel (näiteks ptk 1 näite 6 juhul, kus U ja X ära vahetada, on tegu 2-astme Markovi ahelaga). Kolmekaupa Markovi ahel $(U_t, X_t, Y_t)_{t=1}^n$ on paarikaupa Markovi ahel $(W_t, Y_t)_{t=1}^n$, kus $W_t = (U_t, X_t)$ ja seega saab Viterbi algoritmiga leida $(\hat{u}, \hat{x}) = \arg\max_{x,u} p(x, u|y)$.

Nimetame $\hat{x}$ leidmist **W-Viterbi** algoritmiks. Üldiselt W-Viterbi teel saadud $\hat{x}$ ei ole sama kui $x^* = \arg\max_x p(x|y)$, mida kinnitavad numbrilised tulemused peatükis 3. Kehtivad võrratused

$$|\mathcal{U}|^n p(\hat{x}, \hat{u}|y) \geq p(x^*|y) \geq p(\hat{x}|y) \geq p(\hat{x}, \hat{u}|y) \geq \frac{p(x^*|y)}{|\mathcal{U}|^n} > 0,$$

sest $p(x^*|y) = \sum_{u \in \mathcal{U}^n} p(x^*, u|y) \leq \sum_{u \in \mathcal{U}^n} p(\hat{x}, \hat{u}|y) = |\mathcal{U}|^n p(\hat{x}, \hat{u}|y)$.

Kui kolmekaupa Markovi ahela marginaalprotsessil $(x_t|y)$ ei ole ühegi väikse k korral k-astme Markovi omadust, siis võime loota, et protsessil on mõnda sorti segunemisomadus nii, et "kaugel minevikul on väike mõju tulevikule". Näiteks võime loota, et piisavalt suure k korral kehtib ligikaudne k-astme Markovi omadus

$$p(x_t|x_{t-1}, ..., x_1, y) \approx p(x_t|x_{t-1}, ..., x_{t-k}, y)$$

või kehtib omadus

$$p(x|y) \approx p(x_1, ..., x_k|y)p(x_{k+1}|x_1, ..., x_k, y) \cdots p(x_n|x_{n-k}, ..., x_{n-1}, y).$$

Sellistes olukordades võime oletada, et k-Viterbi algoritm annab ligilähedase tulemuse Viterbi rajale.

Kirjutame eelneva mõttekäigu pikemalt välja. Kui me teame, et

$$\forall x \in \mathcal{X}^n : \quad |p(x|y) - p(x_1, ..., x_k|y) \cdots p(x_n|x_{n-k}, ..., x_{n-1}, y)| \leq \varepsilon$$

ja tähistame

$$V(x) := p(x_1, ..., x_k|y) \cdots p(x_n|x_{n-k}, ..., x_{n-1}, y),$$
$$\hat{x} := \arg\max_x p(x|y), \qquad \overline{x} := \arg\max_x V(x),$$

siis

$$p(\hat{x}|y) - p(\overline{x}|y) \leq 2\varepsilon,$$

sest

$$2\varepsilon \geq p(\hat{x}|y) - V(\hat{x}) + V(\overline{x}) - p(\overline{x}|y) \geq p(\hat{x}|y) - p(\overline{x}|y).$$

Üldisemalt stohhastiliste protsesside lähendamist k-astme Markovi ahelaga on varem uuritud. Näiteks kui stohhastiline protsess on stationaarne ja lõpliku olekute hulgaga,

siis k kasvades protsessi lähendades k-astme Markovi ahelaga lähevad nulli suhteline entroopiamäär [Lem22] ja $\overline{d}$-kaugus [GLT13]. Ka mittestatsionaarsete protsesside puhul saab teatud eelduste puhul näidata, et $\overline{d}$-kaugus läheb nulli [FG02]. Kolmekaupa Markovi ahelate korral vajavad edasist uurimist küsimused:

1. kas k kasvades k-Viterbi algoritmi väljund on mingist hetkest Viterbi rada?
2. kui suur k tuleb valida, et Viterbi raja lähendi viga oleks tolereeritavalt väike?

2.5 k-blokk algoritm

K-blokk algoritmi on varasemalt kasutatud, et paarikaupa Markovi ahelates leida omadustelt PMAP ja Viterbi ahelate vahelisi ahelaid [KL22]. Meie kasutame k-blokk algoritmi, et lähendada Viterbi rada, maksimiseerides funktsiooni

$$p(x^n|y^n)$$

asemel funktsiooni

$$B_k(x) := \sum_{t=2-k}^{n} \ln p(x_{t\vee 1}^{(t+k-1)\wedge n}|y^n). \tag{4}$$

Tähistused $\vee$ ja $\wedge$ tähendavad vastaval maksimumi ja miinimumi võtmist: $a \vee b := \max\{a, b\}$ ja $a \wedge b := \min\{a, b\}$.

Näiteks, kui $k = 3$ ja $n = 5$, siis

$$\begin{aligned}\sum_{t=2-k}^{n} \ln p(x_{t\vee 1}^{(t+k-1)\wedge n}|y^n) &= \ln p(x_1|y^n) + \ln p(x_1, x_2|y^n) \\ + \ln p(x_1, x_2, x_3|y^n) + \ln p(x_2, x_3, x_4|y^n) &+ \ln p(x_3, x_4, x_5|y^n) \\ + \ln p(x_4, x_5|y^n) &+ \ln p(x_5|y^n).\end{aligned}$$

Tähistame k-Viterbi algoritmi sihifunktsiooni logaritmitud kujul järgmiselt:

$$V_k(x) := \ln p(x_1, ..., x_k|y) + \ln p(x_{k+1}|x_1, ..., x_k, y) + \cdots + \ln p(x_n|x_{n-k}, ..., x_{n-1}, y).$$

Lemma 2.1. *[KL22, LK14] K-blokk algoritmi sihifunktsioon on esimese k k-Viterbi sihifunktsiooni summa*

$$B_k(x) = \sum_{i=0}^{k-1} V_i(x).$$

Tõestus. Näeme, et $B_1(x) = V_0(x)$ ning kui $k > 1$, siis

$$\begin{aligned}
B_k(x) - B_{k-1}(x) &= \ln p(x_1, ..., x_{k-1}|y) + \ln p(x_1, ..., x_k|y) + \ln p(x_2, ..., x_{k+1}|y) + ... \\
&\quad + \ln p(x_{n-k+1}, ..., x_n|y) \\
&\quad - \ln p(x_1, ..., x_{k-1}|y) - \ln p(x_2, ..., x_k|y) - ... - \ln p(x_{n-k+1}, ..., x_{n-1}|y) \\
&= \ln p(x_1, ..., x_{k-1}|y) + \ln \frac{p(x_1, ..., x_k|y)}{p(x_1, ..., x_{k-1}|y)} + ... + \ln \frac{p(x_{n-k+1}, ..., x_n|y)}{p(x_{n-k+1}, ..., x_{n-1}|y)} \\
&= \ln p(x_1, ..., x_{k-1}|y) + \ln p(x_k|x_1, ..., x_{k-1}, y) + ... + \ln p(x_n|x_{n-k+1}, ..., x_{n-1}, y) \\
&= V_{k-1}.
\end{aligned}$$

Eelnevas, kui $\ln p(x_1, ..., x_{k-1}|y) = -\infty$, siis ka $\ln p(x_1, ..., x_k|y) = -\infty$ ning $\ln p(x_1^n|y) = -\infty$. Seega praktikas on negatiivsete lõpmatuste vahe $\ln p(x_1, ..., x_k|y) - \ln p(x_1, ..., x_{k-1}|y) = -\infty - (-\infty) = -\infty$.

Järeldus 2.1.1. *k-blokk algoritmi sihifunktsiooni võib vaadelda kui Cesàro summat k-Viterbi algoritmi sihifunktsioonidest*

$$B_k(x) \propto \frac{1}{k} \sum_{i=0}^{k-1} V_i(x).$$

Seega võib motiveerida k-bloki algoritmi kasutamist uskumusega, et Cesàro summal on head omadused, analoogselt kuidas Fourier' ridade puhul Cesàro summat kasutades saab ühtlase koonduvuse (Fejér teoreem)[Lei14, lk 154]. Näiteks kui $n = \lceil \alpha k \rceil$, kus $\alpha \in (1, \infty)$, ja $\lim_{k\to\infty} V_k(x) - \ln p(x|y) = 0$, siis ka $\lim_{k\to\infty} \frac{B_k(x)}{k} - \ln p(x|y) = 0$. Samas kui oletame, et piirväärtust $\lim_{k\to\infty} V_k(x) - \ln p(x|y)$ ei eksisteeri, siis hüpoteetiliselt on võimalik, et $\lim_{k\to\infty} \frac{B_k(x)}{k} - \ln p(x|y) = 0$ – tüüpiline näide matemaatilisest analüüsist on, et jadal $(0, 1, 0, 1, ...)$ on Cesàro mõttes piirväärtus $\frac{1}{2}$, aga tavalist piirväärtust ei eksisteeri.

Siiski fikseeritud n korral on lihtsasti näha, et $V_n = \ln p(x|y)$, aga B_k kohta oskame öelda ainult nii palju, et kui $k > n$, siis $B_k = B_n + (k - n) \ln p(x|y)$ ja seega võime ainult öelda, et piisavalt suure k korral $\arg\max_x B_k(x) \equiv \arg\max_x p(x|y)$, sest hulk $\mathcal{X}$ on lõplik. Artikli [KL22] simulatsioonid näitavad, et pikkusega $n = 1000$ paarikaupa Markovi ahela korral võib k olla suurem kui 9000 enne kui k-blokk algoritmi väljund on Viterbi rada.

Järeldus 2.1.2. *[KL22, LK14] Kui marginaalprotsess (X_t, Y_t) on Markovi ahel ja $k > 1$, siis*

$$B_k(x) - B_{k-1}(x) = \ln p(x|y)$$

ja seega

$$B_k(x) = V_0(x) + (k-1)\ln p(x|y).$$

Sealhulgas, kui $x^{(k)} = \arg\max_x B_k(x)$ ***ja*** $x^{(k-1)} = \arg\max_x B_{k-1}(x)$***, siis*** $p(x^{(k)}|y) \geq p(x^{(k-1)}|y)$.

Tõestus. Esimene võrdus järeldub otse lemmast 2.1 ja asjaolust, et iga Markovi ahel on ka kõrgema astme Markovi ahel. Teine võrdus on otsene järeldus esimesest.

Võrratuse $p(x^{(k)}|y) \geq p(x^{(k-1)}|y)$ saame kasutades esimest võrdust. Näeme, et

$$B_{k-1}(x^{(k)}) + \ln p(x^{(k)}|y) \geq B_{k-1}(x^{(k-1)}) + \ln p(x^{(k-1)}|y)$$

ja seega

$$\ln p(x^{(k)}|y) - \ln p(x^{(k-1)}|y) \geq B_{k-1}(x^{(k-1)}) - B_{k-1}(x^{(k)}) \geq 0.$$

Blokkide tõenäosused saab leida edasi-tagasi algoritmi üldistava k-edasi-tagasi algoritmi abil.

2.5.1 B_k maksimiseerimine

Esitame k-blokk algoritmi analoogselt artiklis [KL22] esitatud Rabiner k-blokk algoritmile.

Iga $a^{k-1} \in \mathcal{X}^{k-1}$ korral leiame

$$\delta_0^k(a^{k-1}) := \ln p\left(x_1 = a_1|y^n\right) + \ln p\left(x_1 = a_1, x_2 = a_2|y^n\right) + ... + \ln p\left(x_1^{k-1} = a^{k-1}|y^n\right)$$

ning lisaks iga $t = 1, ..., n-k+1$ korral

$$\delta_t^k(a^{k-1}) := \max_{x\in\mathcal{X}} \left[\delta_{t-1}^k(x, a^{k-2}) + \ln p\left(x_t = x, x_{t+1}^{t+k-1} = a^{k-1}|y^n\right)\right],$$

$$\gamma_t^k(a^{k-1}) := \arg\max_{x\in\mathcal{X}} \left[\delta_{t-1}^k(x, a^{k-2}) + \ln p\left(x_t = x, x_{t+1}^{t+k-1} = a^{k-1}|y^n\right)\right].$$

Viimaks leiame

$$\delta_{n-k+2}^k(a^{k-1}) := \delta_{n-k+1}^k(a^{k-1}) + \ln P\left(X_{n-k+2}^n = a^{k-1}|y^n\right) + \ln P\left(X_{n-k+3}^n = a_2^{k-1}|y^n\right) + ... \\ ... + \ln P\left(X_n = a_{k-1}|y^n\right).$$

Paneme tähele, et sihifunktsiooni B_k maksimum on võrdne maksimumiga $\max_{a^{k-1}} \delta^k_{n-k+2}(a^{k-1})$. Sihifunktsiooni B_k maksimiseeriva x^n leiame nüüd tagasi liikumise teel:

$$x^n_{n-k+2} = \arg\max_{a^{k-1}} \delta^k_{n-k+2}(a^{k-1})$$

ja iga $t = 1, ..., n-k+1$ korral

$$x_t = \gamma^k_t(x^{t+k-1}_{t+1}).$$

Analoogselt saab leida $(k-1)$-Viterbi sihifunktsiooni maksimumi, kasutades ära asjaolu, et $V_{k-1}(x^n) = B_k(x^n) - B_{k-1}(x^n)$. Võtame

$$\delta^k_0(a^{k-1}) := 0,$$

iga $t = 1, ..., n-k+1$ korral

$$\delta^k_t(a^{k-1}) := \max_{x\in\mathcal{X}}\big[\delta^k_{t-1}(x, a^{k-2}) + \ln p\left(x_t = x, x^{t+k-1}_{t+1} = a^{k-1}|y^n\right) - \ln p\left(x^{t+k-1}_{t+1} = a^{k-1}|y^n\right)\big],$$

$$\gamma^k_t(a^{k-1}) := \arg\max_{x\in\mathcal{X}}\big[\delta^k_{t-1}(x, a^{k-2}) + \ln p\left(x_t = x, x^{t+k-1}_{t+1} = a^{k-1}|y^n\right) - \ln p\left(x^{t+k-1}_{t+1} = a^{k-1}|y^n\right)\big]$$

ning

$$\delta^k_{n-k+1}(a^{k-1}) := \max_{x\in\mathcal{X}}\big[\delta^k_{n-k}(x, a^{k-2}) + \ln p\left(x_{n-k+1} = x, x^n_{n-k+2} = a^{k-1}|y^n\right)\big],$$

$$\gamma^k_t(a^{k-1}) := \arg\max_{x\in\mathcal{X}}\big[\delta^k_{n-k}(x, a^{k-2}) + \ln p\left(x_{n-k+1} = x, x^n_{n-k+2} = a^{k-1}|y^n\right)\big],$$

kus praktilistel kaalutlustel $\ln 0 - \ln 0 = -\infty$. Sihifunktsiooni maksimiseeriva x^n leiame tagasi liikumise teel:

$$x^n_{n-k+2} = \arg\max_{a^{k-1}} \delta^k_{n-k+1}(a^{k-1})$$

ja iga $t = 1, ..., n-k+1$ korral

$$x_t = \gamma^k_t(x^{t+k-1}_{t+1}).$$

Kui k-edasi-tagasi algoritmi abil on blokkide (Log) tihedused leitud, siis nii $(k-1)$-Viterbi ja k-blokk algoritmide ajaline keerukus on $O(n|\mathcal{X}|^k)$. Kuna mõlema algoritmi jaoks on vaja kasutada k-edasi-tagasi algoritmi, mille ajaliseks keerukuseks on $O(nk|\mathcal{U}\times\mathcal{X}|^k)$, siis loeme nende algoritmide keerukuseks k-edasi-tagasi algoritmi keerukust.

2.6 Segmenteerimis EM-algoritm (SEM)

Artiklis [LGKK19] vahetati traditsioonilises EM-algoritmis parameetri θ ja varitunnuse X^n rollid, et varjatud Markovi mudelites maksimiseerida tihedust $p(x^n|y^n)$, kus alg-jaotust ja üleminekutihedusi määrav mudeli parameeter θ on juhuslik suurus. Mainitud artikli autorid nimetasid seda segmenteerimis EM-algoritmiks. Sellest inspireerituna defineerime me oma segmenteerimis EM-algoritmi, lühidalt SEM-i, asendades parameetri θ varitunnusega U^n.

Olgu meil esialgne hinnang Viterbi rajale $x^{(0)} \in \mathcal{X}^n$. Segmenteerimis EM-algoritmis leiame varasema hinnangu $x^{(i)}$ põhjal hinnangu $x^{(i+1)}$ järgmiselt:

$$\begin{aligned} x^{(i+1)} &= \arg\max_{x \in \mathcal{X}^n} \left[\sum_{u \in \mathcal{U}^n} \ln p(u, x|y) p(u|x^{(i)}, y) \right] \\ &= \arg\max_{x \in \mathcal{X}^n} \left[\sum_{u \in \mathcal{U}^n} \ln p(x, y|u) p(u|x^{(i)}, y) \right]. \end{aligned}$$

Sealhulgas kehtib võrratus

$$p(x^{(i+1)}|y) \geq p(x^{(i)}|y).$$

Tõestus. Võrratuse tõestus on identne standartse EM algoritmi tõestusega [Wik23b], kus näidatakse, et $p(y|x^{(i+1)}) \geq p(y|x^{(i)})$.

Iga $u \in \mathcal{U}^n$ korral, mille tihedus $p(u|x, y)$ on nullist suurem, saame kirjutada

$$\ln p(x|y) = \ln p(u, x|y) - \ln p(u|x, y).$$

Võttes võrduse mõlemalt poolt tingliku keskväärtuse üle varitunnuse u saame, et

$$\begin{aligned} \ln p(x|y) &= \sum_{u \in \mathcal{U}^n} \ln p(x, u|y) p(u|y, x^{(i)}) - \sum_{u \in \mathcal{U}^n} \ln p(u|y, x) p(u|y, x^{(i)}) \\ &= Q(x|x^{(i)}) + H(x|x^{(i)}), \end{aligned}$$

kus Q ja H asendavad vastavat summat ja negatiivset summat. Moodustame vahe

$$\ln p(x|y) - \ln p(x^{(i)}|y) = Q(x|x^{(i)}) - Q(x^{(i)}|x^{(i)}) + H(x|x^{(i)}) - H(x^{(i)}|x^{(i)}).$$

Gibbsi võrratus ütleb, et $H(x|x^{(i)}) - H(x^{(i)}|x^{(i)}) \geq 0$, millest järeldub, et

$$\ln p(x|y) - \ln p(x^{(i)}|y) \geq Q(x|x^{(i)}) - Q(x^{(i)}|x^{(i)}).$$

Paneme tähele, et SEM-algoritmis maksimiseerime funktsiooni $Q(x|x^{(i)})$. Viimasest võrratusest järeldub, et

$$\ln p(x^{(i+1)}|y) \geq \ln p(x^{(i)}|y) + \max_x Q(x|x^{(i)}) - Q(x^{(i)}|x^{(i)}),$$

millest järeldub tõestatav võrratus.

2.6.1 SEM maksimumi leidmine

Eesmärk on maksimiseerida funktsiooni

$$\sum_u \ln p(u,x|y)p(u|x^{(i)},y). \tag{5}$$

Paneme tähele, et $p(u,x|y) = \frac{p(z)}{p(y)} \propto p(z) = p(z_1)\prod_{t=2}^{n} p(z_t|z_{t-1})$ ning tähistame

$$q^{(i)}(u) := p(u|x^{(i)},y), \quad q_t^{(i)}(a) := \sum_{u:\, u_t=a} p(u|x^{(i)},y), \quad q_{t-1,t}^{(i)}(a,b) := \sum_{\substack{u:\\ u_{t-1}=a,\\ u_t=b}} p(u|x^{(i)},y).$$

Nüüd funktsiooni (5) maksimiseerimine on samaväärne järgmise funktsiooni maksimiseerimisega:

$$\begin{aligned}\sum_u \ln p(z)q^{(i)}(u) &= \sum_u \left[\ln p(z_1)q^{(i)}(u) + \sum_{t=2}^{n} \ln p(z_t|z_{t-1})q^{(i)}(u)\right] \\ &= \sum_a \ln p(z_1)q_1^{(i)}(a) + \sum_{t=2}^{n}\sum_{a,b} \ln p(z_t|z_{t-1})q_{t-1,t}^{(i)}(a,b).\end{aligned}$$

Kuna

$$q_t^{(i)}(a) = p(u_t = a|x^{(i)},y) = \frac{p(u_t=a,x^{(i)},y)}{p(x^{(i)},y)}$$

ja

$$q_{t-1,t}^{(i)}(a,b) = p(u_{t-1}=a,u_t=b|x^{(i)},y) = \frac{p(u_{t-1}=a,u_t=b,x^{(i)},y)}{p(x^{(i)},y)},$$

saame maksimiseeritava funktsiooni viia kujule

$$\sum_a \ln p(z_1)p(u_t=a,x^{(i)},y) + \sum_{t=2}^{n}\sum_{a,b} \ln p(z_t|z_{t-1})p(u_{t-1}=a,u_t=b,x^{(i)},y), \tag{6}$$

kus $p(u_t=a,x^{(i)},y)$ ja $p(u_{t-1}=a,u_t=b,x^{(i)},y)$ on leitavad edasi-tagasi ja 2-edasi-tagasi algoritmiga.

Maksimiseerimisülesande saab nüüd lahendada dünaamilise planeerimisega, analoogselt 2-blokk algoritmile.

Tähistame

$$f_1(r) = \sum_a \ln p(u_1 = a, x_1 = r, y) p(u_t = a, x^{(i)}, y)$$

ja

$$f_{t-1,t}(r,s) = \sum_{a,b} \ln p(u_t = b, x_t = s, y_t | u_{t-1} = a, x_{t-1} = r, y_{t-1}) p(u_{t-1} = a, u_t = b, x^{(i)}, y).$$

Maksimum on leitav järgmiselt. Leiame

$$\delta_1(r) := f_1(r)$$

ja iga sammu $t = 2, ..., n$ korral

$$\delta_t(s) := \max_r(\delta_{t-1}(r) + f_{t-1,t}(r,s)),$$

$$\gamma_t(s) := \arg\max_r(\delta_{t-1}(r) + f_{t-1,t}(r,s)).$$

Paneme tähele, et funktsiooni (6) maksimum on sama kui $\max_s \delta_n(s)$ ning funktsiooni maksimiseerija $x^{(i+1)}$ on leitav tagasiliikumisega:

$$x_n^{(i+1)} = \arg\max_s \delta_n(s)$$

ja iga $t = n-1, n-2, ..., 1$ korral

$$x_t^{(i+1)} = \gamma_{t+1}(x_{t+1}^{(i+1)}).$$

Algoritmi keerukus on $O\left(n|\mathcal{U} \times \mathcal{X}|^2\right)$.

2.7 Mäkke ronimise algoritm (HC)

Mäkke ronimine on optimiseerimismeetod, kus mitme muutujaga optimiseerimisülesandes leitakse esialgse lahendi lähendist parem lähend ühe muutuja kaupa [Wik23c]. Olgu meil esialgne hinnang Viterbi rajale ${}^{(0)}x$. Mäkke ronimise algoritmiga leiame järgmise hinnangu ${}^{(i+1)}x$, tehes eelnevas hinnangus ${}^{(i)}x$ väikese muudatuse, leides $k \in \{1, ..., n\}$ ja $a \in \mathcal{X}$ nii, et tihedus

$$p\left({}^{(i+1)}x_1^{k-1} = {}^{(i)}x_1^{k-1},\ {}^{(i+1)}x_k = a, {}^{(i+1)}x_{k+1}^{n} = {}^{(i)}x_{k+1}^{n},\ y\right) \tag{7}$$

on maksimaalne.

Näiteks, kui ahela pikkus on $n = 3$, tähestik on $\mathcal{X} = \{a, b, c\}$ ja esialgne hinnang Viterbi rajale on ${}^{(0)}x = aab$, siis valime ${}^{(1)}x$ potentsiaalsete kandidaatide

aab	bab	abb	aac
	cab	acb	aaa

seast nii, et tõenäosus $P({}^{(1)}x, y)$ on kõige suurem.

Muudatuste tõenäosused leiame edasi-tagasi algoritmiga. Esialgu leiame

$$\alpha_t^{(i)}(u_t) := p(u_t, {}^{(i)}x_1^t, y_1^t) = \sum_{u_{t-1}} p(u_t, {}^{(i)}x_t, y_t | u_{t-1}, {}^{(i)}x_{t-1}, y_{t-1}) \alpha_{t-1}^{(i)}(u_{t-1})$$

ja

$$\beta_t^{(i)}(u_t) := p({}^{(i)}x_{t+1}^n, y_{t+1}^n | u_t, {}^{(i)}x_t, y_t) = \sum_{u_{t+1}} p(u_{t+1}, {}^{(i)}x_{t+1}, y_{t+1} | u_t, {}^{(i)}x_t, y_t) \beta_{t+1}^{(i)}(u_{t+1}).$$

Nüüd iga $a \in \mathcal{X}$ korral leiame

$${}^*\alpha_t^{(i)}(u_t, a) := \sum_{u_{t-1}} p(u_t, x_t = a, y_t | u_{t-1}, {}^{(i)}x_{t-1}, y_{t-1}) \alpha_{t-1}^{(i)}(u_{t-1})$$

ja

$${}^*\beta_t^{(i)}(u_t, a) := \sum_{u_{t+1}} p(u_{t+1}, {}^{(i)}x_{t+1}, y_{t+1} | u_t, x_t = a, y_t) \beta_{t+1}^{(i)}(u_{t+1}).$$

Tõenäosus (7) on leitav järgmiselt

$$\sum_{u_k} {}^*\alpha_t^{(i)}(u_k, a)\, {}^*\beta_t^{(i)}(u_k, a).$$

Algoritmi ühe sammu ajaline keerukus on $O(n|\mathcal{X}||\mathcal{U}|^2)$.

Mäkke ronimise algoritm on sarnane artiklis [LGKK19] kasutatud ICM algoritmiga, aga erinevus seisneb selles, et HC algoritmis valitakse indeks, milles tehakse muudatus nii, et tiheduse väärtuse kasv oleks kõige suurem, aga ICM algoritmis valitakse indekse järjest vasakult paremale. Seega võib öelda, et HC algoritm on ahnem.

HC algoritmil on ka loomulik üldistus, kus muudetakse mitut indeksit korraga. See üldistus on üldiselt ebapraktiline, sest näiteks kui vaadata kahte indeksit korraga, siis ajaline keerukus on vähemalt $n(n-1)$.

3 Eksperimendid

Illustreerime erinevaid peatükis 2 tutvustatuid lähendusalgoritme kolmekaupa Markovi ahelatel Viterbi raja $\arg\max_{x\in\mathcal{X}^n}\sum_{u\in\mathcal{U}^n} p(x,u,y)$ leidmiseks, rakendades neid erinevatel näidetel. Võrdleme omavahel k-Viterbi ja k-blokk algoritmi, uurides nende algoritmide sooritusvõimet ja monotoonsust k kasvades. Piirdume k väärtustega, mis jäävad alla seitsme, sest algoritmide teostusesd võtavad suuremate k väärtuste korral palju operatiivmälu ning aega. Veel võrdleme omavahel iteratiivseid meetodeid, segmenteerimis EM (SEM) ja mäkke ronimise (HC) algoritmi, kus etteantud alglähendist saadakse jada monotoonselt paremaid lähendeid. Jooksutame iteratiivseid algoritme kuni algoritm jõuab püsipunkti - lähendini, millele algoritmi rakendades jääb lähend samaks. Hindame iteratiivsete algoritmide sooritusvõimet ning püsipunkti jõudmise sammude arvu. Demonstreerime, et SEM ja HC algoritmide püsipunktid võivad erineda ja seega võib praktikas olla mõistlikum kasutada segualgoritmi, rakendades SEM ja HC algoritmi vaheldumisi.

Tabel 1: Algoritmide ajalised keerukused.

Algoritm	ajaline keerukus
W-Viterbi	$O(n\|\mathcal{U}\times\mathcal{X}\|^2)$
0-Viterbi	$O(n\|\mathcal{U}\times\mathcal{X}\|^2)$
(k-1)-Viterbi	$O(nk\|\mathcal{U}\times\mathcal{X}\|^k)$
k-blokk	$O(nk\|\mathcal{U}\times\mathcal{X}\|^k)$
SEM	$O(n\|\mathcal{U}\times\mathcal{X}\|^2)$
HC	$O(n\|\mathcal{X}\|\|\mathcal{U}\|^2)$

Algoritmide sooritusvõimet hindame Log tiheduste vahena. Kui x^* on kõikide uuritavate lähendite seast valitud parim lähend ja $\hat{x}$ on ühest algoritmist saadud lähend, siis nende Log tiheduste vahe on

$$\text{LV} = \ln p(\hat{x}|y) - \ln p(x^*|y) = \ln\left(\frac{p(\hat{x}|y)}{p(x^*|y)}\right) = \ln\left(1+\frac{p(\hat{x}|y)-p(x^*|y)}{p(x^*|y)}\right).$$

Viimasest võrdusest on näha LV seost suhtelise veaga $\frac{p(\hat{x}|y)-p(x^*|y)}{p(x^*|y)}$.

3.1 Diskreetsed mudelid

Diskreetseteks mudeliteks peame kolmekaupa Markovi ahelaid, kus lisaks olekute hulkadele $\mathcal{X}$ ja $\mathcal{U}$ on ka olekute hulk $\mathcal{Y}$ lõplik. Tuletame meelde, et

$$\begin{aligned} p(z_{t+1}|z_t) &= p(u_{t+1},x_{t+1},y_{t+1}|u_t,x_t,y_t) = \\ &= p(y_{t+1}|u_{t+1},x_{t+1},u_t,x_t,y_t)p(x_{t+1}|u_{t+1},u_t,x_t,y_t)p(u_{t+1}|u_t,x_t,y_t). \end{aligned}$$

Mudelid konstrueerime üleminekumaatriksite

$$[D_{ij}] = p(z_{t+1} = j | z_t = i)$$

või

$$\begin{aligned}
[A_{ij}](x_t, y_t) &= p(u_{t+1} = j | u_t = i, x_t, y_t), \\
[B_{ij}](u_{t+1}, u_t, y_t) &= p(x_{t+1} = j | u_{t+1}, u_t, x_t = i, y_t), \\
[C_{ij}](u_{t+1}, x_{t+1}, u_t, x_t) &= p(y_{t+1} = j | u_{t+1}, x_{t+1}, u_t, x_t, y_t = i)
\end{aligned}$$

abil.

Kui konstrueerime maatriksid juhuslikult, siis kasutame selle jaoks Dirichlet' jaotust, sest see annab ühtlase jaotuse üleminekumaatriksite hulgal [Cha10]. Täpsemalt kui $[M_{ij}]$ on maatriks, siis iga i korral sõltumatult $(M_{i1}, M_{i2}, ..., M_{im}) \sim \text{Dir}(\underbrace{1, 1, ..., 1}_{m})$, kus m on maatriksi rea pikkus. Selle jaotuse tihedusfunktsioon on [Wik23a]

$$\begin{cases} \frac{1}{(m-1)!}, \text{ kui } \sum_{j=1}^{m} M_{ij} = 1 \text{ ja } \forall j : M_{ij} \in [0, 1], \\ 0. \end{cases}$$

Markovi ahela algoleku Z_1 valime ühtlase jaotusega.

3.1.1 Üldine TMM(2,2,2)

Olgu olekute hulgad $\mathcal{U} = \{0, 1\}, \mathcal{X} = \{0, 1\}, \mathcal{Y} = \{0, 1\}$ ning 8×8 üleminekumaatriks $[D_{ij}] = p(z_{t+1} = j | z_t = i)$ on saadud Dirichlet' jaotuse abil. Genereerisime 100 üleminekumaatriksit ning iga maatriksi korral 100 realisatsiooni. Iga realisatsiooni ja valitud algoritmi korral leiame lähendi $\hat{x}$ kaudu Log tiheduse $\ln p(\hat{x}|y)$ ning Log tiheduste vahe LV $= \ln p(\hat{x}|y) - \ln p(x^*|y)$, kus x^* on kõikide uuritud algoritmide väljundite seast parim. Tabelisse kandsime väärtuste $\ln p(\hat{x}|y)$ ja LV keskmise üle kõigi 10000 realisatsiooni. Tabelis 2 on esimese nelja k-bloki ning neile vastava k-Viterbi algoritmi tulemused. Lisaks on ära toodud võrdluseks realisatsioonis olnud tegeliku ahela (tabelis kannab nime varjatud), W-Viterbi ning 0-Viterbi tulemused. Näeme, et k-Viterbi algoritm toimib tabelis 2 toodud algoritmidest üldiselt kõige paremini.

Tabel 2: Tulemused.

Algoritm	$\ln p(\hat{x}\|y)$	LV
varjatud	-128.27	$2.076 \cdot 10$
W-Viterbi	-109.98	2.472
0-Viterbi	-114.02	6.515

Algoritm	$\ln p(\hat{x}\|y)$	LV
1-Viterbi	-108.47	$9.621 \cdot 10^{-1}$
2-Viterbi	-107.62	$1.125 \cdot 10^{-1}$
3-Viterbi	-107.52	$1.254 \cdot 10^{-2}$
4-Viterbi	-107.51	$7.826 \cdot 10^{-4}$

Algoritm	$\ln p(\hat{x}\|y)$	LV
2-Blokk	-109.26	1.754
3-Blokk	-108.15	$6.423 \cdot 10^{-1}$
4-Blokk	-107.83	$3.248 \cdot 10^{-1}$
5-Blokk	-107.71	$2.022 \cdot 10^{-1}$

K-Viterbi ja k-blokk algoritmide käitumise uurimiseks k kasvades uurime nende algoritmide monotoonsust, sorteerides iga realisatsiooni korral k-Viterbi algoritmidest saadud lähendite Log tihedused ning märkides ära sorteerimise järjekorra. Näiteks kui $\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4$ on vastavalt 1-Viterbi, 2-Viterbi, 3-Viterbi ja 4-Viterbi algoritmidest saadud lähendid ning $p_1 = \ln p(\hat{x}_1|y)$, $p_2 = \ln p(\hat{x}_2|y)$, $p_3 = \ln p(\hat{x}_3|y)$ ja $p_4 = \ln p(\hat{x}_4|y)$, siis järjestikusele $p_1 < p_2 < p_3 < p_4$ vastab permutatsioon 0 1 2 3 ning järjestikusele $p_3 < p_1 < p_4 < p_2$ vastab permutatsioon 2 0 3 1[2]. Tabelis 3 on kõik sellised empiiriliselt kogutud k-Viterbi permutatsioonid kokku loendatud ning analoogne loetelu on antud ka k-Blokk algoritmi jaoks. Mainime ära, et k-Viterbi algoritmi korral esines kõiki permutatsioone, välja arvatud 3 2 1 0, 3 0 2 1 ja 3 2 0 1. Näeme, et k-Viterbi algoritmi korral oli ligikaudu $1/10$ kordadel monotoonsus rikutud ja k-blokk algoritmil oli tunduvalt monotoonsem – ligikaudu $1/100$ juhtumit rikkusid monotoonsust.

Iteratiivsete algoritmide uurimiseks valisime madala ajalise keerukusega algoritmid: W-Viterbi, 0-Viterbi, 1-Viterbi ning 2-Blokk, et saada esialgne hinnang Viterbi rajale. Esialgsele hinnangule rakendasime segmenteerimis EM-algoritmi (SEM), mäkke ronimise algoritmi (HC) kuni iteratiivsed algoritmid jõudsid püsipunkti. Et demonstreerida SEM ja HC algoritmide püsipunktide erinevust vaatame, mis juhtub kui SEM algoritmile rakendada otsa HC algoritmi, tähistades seda HCSEM, ning kui HC algoritmile rakendada otsa SEM algoritmi, mida tähistame SEMHC. Tabelis 4 on näha iteratiivsete algoritmide tulemusi. Näeme, et valitud alglähendite korral nii SEM kui ka HC algoritm ei paku üldiselt olulist võitu. Tabelist 5 näeb, et valitud alglähendite korra teevad iteratsiooni algoritmid keskmiselt vähe samme. Võib väita, et SEM algoritmi puhul

[2]Märgime, et kui järjestikuses $p_1 < p_2 < p_3 < p_4$ mõne võrratuse asemel on võrdus, siis permutatsioon on jätkuvalt 0 1 2 3, aga kui mõnes teise järjestikus esineb võrdus, siis sõltub see sorteerimisalgoritmi eripäradest. Näiteks $p_2 = p_4 < p_1 < p_3$ olla nii permutatsioon 1 3 0 2 kui ka permutatsioon 3 1 0 2.

Tabel 3: k-Viterbi ja k-Blokk monotoonsus

k-Viterbi	
Permutatsioon	kogus
0 1 2 3	8856
1 0 2 3	341
0 2 1 3	284
0 1 3 2	228
0 2 3 1	68
0 3 1 2	49
1 0 2 3	45
ülejäänud	129

k-Blokk	
Permutatsioon	kogus
0 1 2 3	9893
1 0 2 3	59
1 2 0 3	19
0 2 1 3	14
1 2 3 0	10
0 2 3 1	2
0 1 3 2	2
2 3 0 1	1

Tabel 4: Iteratiivsete algoritmide tulemused

Algoritm	W-Viterbi		0-Viterbi		1-Viterbi		2-Blokk	
	$\ln p(\hat{x}\|y)$	LV	$\ln p(\hat{x}\|y)$	LV	$\ln p(\hat{x}\|y)$	LV	$\ln p(\hat{x}\|y)$	LV
esialgne	-109.98	2.472	-114.02	6.515	-108.47	$9.621 \cdot 10^{-1}$	-109.26	1.754
SEM	-109.80	2.288	-112.12	4.615	-108.42	$9.152 \cdot 10^{-1}$	-109.19	1.682
HC	-109.97	2.459	-112.39	4.880	-108.43	$9.179 \cdot 10^{-1}$	-109.15	1.643
HCSEM	-109.82	2.308	-111.37	3.861	-108.40	$8.942 \cdot 10^{-1}$	-109.10	1.593
SEMHC	-109.76	2.253	-111.43	3.919	-108.39	$8.783 \cdot 10^{-1}$	-109.09	1.580

on juba valitud alglähendid enamasti püsipunktid. Kuna tabelis 4 HCSEM ja SEMHC keskmine Log tihedus erineb SEM ja HC omast, siis saame väita, et HC ja SEM algoritmi püsipunktid võivad erineda.

Tabel 5: Keskmine iteratsioonide arv

Algoritm	W-Viterbi	0-Viterbi	1-Viterbi	2-Blokk
SEM	0.164	0.654	0.045	0.081
HC	0.563	2.028	0.461	0.587

Kuna W-Viterbi, 0-Viterbi, 1-Viterbi ja 2-blokk algrotimide väljundid võivad olla liiga sarnased, siis uurime veel iteratiivsete algoritmide sooritusvõimet juhuslikult genereereritud alglähenditele. Esialgse hinnangu genereerime juhusliku suuruse $(U, X|Y)$ realisatsioonidena. Kuna protsess $(U, X|Y)$ on Markovi ahel, siis saab selle realisatsiooni genereerida üleminekutiheduste kaudu, mis on leitavad edasi-tagasi algoritmi abil

$$p(u_{t+1}, x_{t+1}|u_t, x_t, y) = \frac{p(u_{t+1}, x_{t+1}, u_t, x_t, y)}{p(u_t, x_t, y)}.$$

Genereerisime 10 üleminekumaatriksit ning iga maatriksi korral 50 realisatsiooni ning iga realisatsiooni korral 50 juhuslikult genereeritud alglähendit. Tabelis 6 on näha simulatsioonist saadud numbrilisi väärtusi. Alglähendite keskmine näitab keskmisi väärtusi üle kõigi juhuslikult genereeritud alglähendite. Maksimaalne alglähend näitab keskmisi väärtusi üle kõikide realisatsioonide, kus iga realisatsiooni korral on valitud minimaalse LV väärtusega alglähend. Maksimaalse SEM korral rakendame SEM algoritmi igale alglähendile, valime iga realisatsiooni korral parima SEM väljundi ning võtame keskmise üle kõikide realisatsioonide. Maksimaalne HC on sama kui maksimaalne SEM, aga SEM asemel kasutame HC algoritmi. LV arvutamisel valime x^* 5-Viterbi ja 6-Viterbi algoritmide väljundite seast.

Tabel 6: Iteratiivsete algoritmide tulemused juhuslikel alglähenditel

	$\ln p(\hat{x}\|y)$	LV
alglähendite keskmine	-128.73	21.39
maksmimaalne alglähend	-120.57	13.23
maksimaalne SEM	-113.43	6.09
maksimaalne HC	-117.05	9.71

Näeme, et valides ainult viiskümmend juhuslikku alglähendit ei suuda iteratiivsed algoritmid keskmiselt k-Viterbi algoritmiga konkureerida.

3.1.2 TMM(3,3,3)

Vaatame nüüd mudelit, kus on rohkem olekuid, aga vähem sõltuvusi. Olgu olekute hulgad $\mathcal{U} = \{0, 1, 2\}, \mathcal{X} = \{0, 1, 2\}, \mathcal{Y} = \{0, 1, 2\}$ ning üleminekumaatriksid

$$\begin{aligned}[A_{ij}] &= p(u_{t+1} = j | u_t = i),\\ [B_{ij}](u_{t+1}) &= p(x_{t+1} = j | u_{t+1}, x_t = i),\\ [C_{ij}](u_{t+1}, x_{t+1}) &= p(y_{t+1} = j | u_{t+1}, x_{t+1})\end{aligned}$$

olgu saadud Dirichlet' jaotuse abil. Genereerisime 100 üleminekumaatriksite komplekti ning iga maatriksite komplekti korral genereerisime 100 Markovi ahela realisatsiooni. Loome samasugused tabelid nagu peatükis 3.1.1. Näeme, et saadud tulemused ei erine oluliselt peatüki 3.1.1 omadest.

Tabel 7: Tulemused.

Algoritm	$\ln p(\hat{x}\|y)$	LV
varjatud	-194.40	$3.645 \cdot 10$
W-Viterbi	-160.67	2.720
0-Viterbi	-168.00	$1.01 \cdot 10^{-1}$

Algoritm	$\ln p(\hat{x}\|y)$	LV
1-Viterbi	-158.62	$6.719 \cdot 10^{-1}$
2-Viterbi	-158.08	$1.230 \cdot 10^{-1}$
3-Viterbi	-157.98	$2.435 \cdot 10^{-2}$
4-Viterbi	-157.95	$1.988 \cdot 10^{-3}$

Algoritm	$\ln p(\hat{x}\|y)$	LV
2-Blokk	-159.82	1.869
3-Blokk	-158.78	$8.268 \cdot 10^{-1}$
4-Blokk	-158.41	$4.578 \cdot 10^{-1}$
5-Blokk	-158.25	$2.971 \cdot 10^{-1}$

Tabeli 11 jaoks genereerisime eraldi 10 üleminekumaatriksit ning iga maatriksi korral 50 realisatsiooni ning iga realisatsiooni korral 50 juhuslikult genereeritud alglähendit. Kuna olekute hulgad $\mathcal{U}$ ja $\mathcal{X}$ olid selle mudeli korral suuremad kui 3.1.1 korral, siis arvuti operatiivmälu kokkuhoiu eesmärgil piirdusime LV arvutamisel 4-Viterbi ja 5-Viterbi kasutamisega.

Tabel 8: k-Viterbi ja k-Blokk monotoonsus

k-Viterbi		k-Blokk	
Permutatsioon	kogus	Permutatsioon	kogus
0 1 2 3	9159	0 1 2 3	9955
1 0 2 3	269	1 0 2 3	25
0 2 1 3	253	0 2 1 3	7
0 1 3 2	148	0 1 3 2	5
0 3 1 2	36	0 2 3 1	4
0 2 3 1	35	1 2 0 3	2
1 2 0 3	33	1 2 3 0	1
ülejäänud	67	2 3 0 1	1

Tabel 9: Iteratiivsete algoritmide tulemused

Algoritm	W-Viterbi		0-Viterbi		1-Viterbi		2-Blokk	
	$\ln p(\hat{x}\|y)$	LV	$\ln p(\hat{x}\|y)$	LV	$\ln p(\hat{x}\|y)$	LV	$\ln p(\hat{x}\|y)$	LV
esialgne	-160.67	2.720	-168.00	$1.01 \cdot 10^{-1}$	-158.62	$6.719 \cdot 10^{-1}$	-159.82	1.869
SEM	-160.37	2.413	-162.81	4.853	-158.61	$6.559 \cdot 10^{-1}$	-159.55	1.598
HC	-160.63	2.674	-165.06	7.107	-158.61	$6.577 \cdot 10^{-1}$	-159.74	1.786
HCSEM	-160.35	2.393	-162.10	4.150	-158.60	$6.450 \cdot 10^{-1}$	-159.50	1.547
SEMHC	-160.33	2.375	-162.01	4.062	-158.59	$6.350 \cdot 10^{-1}$	-159.48	1.528

Tabel 10: Keskmine iteratsioonide arv

Algoritm	W-Viterbi	0-Viterbi	1-Viterbi	2-Blokk
SEM	0.385	1.060	0.033	0.352
HC	0.279	2.508	0.137	0.287

Tabel 11: Iteratiivsete algoritmide tulemused juhuslikel alglähenditel

	$\ln p(\hat{x}\|y)$	LV
alglähendite keskmine	-194.04	35.95
maksmimaalne alglähend	-181.70	23.60
maksimaalne SEM	-166.28	8.18
maksimaalne HC	-174.24	16.14

3.2 Mittediskreetne mudel TMM(2,2,$\mathbb{R}$)

Selles alampeatükis vaatame ühte kolmekaupa Markovi ahela mudelit, kus olekute hulgad $\mathcal{U} = \{0, 1, 2\}$ ja $\mathcal{X} = \{0, 1\}$ on lõplikud ning hulk $\mathcal{Y} = \mathbb{R}$ on lõpmatu. Olgu

$$[A_{ij}] = p(u_{t+1} = j | u_t = i),$$
$$[B_{ij}](u_{t+1}) = p(x_{t+1} = j | u_{t+1}, x_t = i)$$

Dirichlet' jaotuse abil saadud üleminekumaatriksid ning iga $t = 2, ..., n$ korral

$$Y_t = X_t + \xi_t,$$

kus $\xi_2, \xi_3, ...$ on iid normaaljaotusega $\mathcal{N}(0, 1.25)$ juhuslikud suurused, mille standardhälve on $\sigma = 1.25$.

Markovi ahela algolekus valime (u_1, x_1) ühtlasest jaotusest ning fikseerime $y_1 = 0$.

Väikese kõrvalepõikena, kirjeldame sellise protsessi Markovi tuuma. Saame väita, et eelmainitud kolmekaupa Markovi mudeli σ-algebra $\mathcal{B}(\mathcal{Z})$ element esitub kujul $\bigcup_{(i,j)\in\mathcal{U}\times\mathcal{X}} \{(i, j)\} \times S_{ij}$, kus $S_{ij} \in \mathcal{B}(\mathbb{R})$ [Ava21, lk 8].

Olgu $C := \bigcup_{(i,j)\in\mathcal{U}\times\mathcal{X}} \{(i, j)\} \times S_{ij} \in \mathcal{B}(\mathcal{Z})$, siis Markovi tuum on

$$\kappa(z_t, C) = P(Z_{t+1} \in C | z_t) = P\left(Z_{t+1} \in \bigcup_{(i,j)\in\mathcal{U}\times\mathcal{X}} \{(i, j)\} \times S_{ij} \,\middle|\, z_t \right)$$
$$= \sum_{i,j} P(U_{t+1} = i, X_{t+1} = j, Y_{t+1} \in S_{ij} | z_t)$$
$$= \sum_{i,j} P(Y_{t+1} \in S_{ij} | U_{t+1} = i, X_{t+1} = j, z_t) P(X_{t+1} = j | U_{t+1} = i, z_t) P(U_{t+1} = i | z_t)$$
$$= \sum_{i,j} P(j + \xi_{t+1} \in S_{ij}) B_{x_t,j}(u_{t+1} = i) A_{u_t,i}.$$

Moodustame nüüd samasugused tabelid nagu peatükis 3.1.1 ja 3.1.2, aga lisaks vaatame sellele mudelile spetsiifilist algoritmi, määrates igale vaatlusele y_t lähima x_t ehk iga t korral leiame $\arg\min_{x_t} |y_t - x_t|$. Kuigi see algoritm on väga kiire, näeme, et selle sooritusvõime jääb alla teistele algoritmidele. Tabelites 12–14 on tulemused sarnased peatüki 3.1.1 ja 3.1.2 omadele.

Tabeli 16 moodutasime sama protseduuriga nagu peatükis 3.1.2 tabeli 11. Erinevalt eelmistest peatükkidest näeme SEM algoritmi puhul oluliselt paremat sooritusvõimet.

Tabel 12: Tulemused.

Algoritm	$\ln p(\hat{x}\|y)$	LV
varjatud	-214.94	$2.141 \cdot 10$
W-Viterbi	-194.79	1.256
0-Viterbi	-197.23	3.693
$\min_{x_t} \|y_t - x_t\|$	-219.93	$2.640 \cdot 10$

Algoritm	$\ln p(\hat{x}\|y)$	LV
1-Viterbi	-193.78	$2.409 \cdot 10^{-1}$
2-Viterbi	-193.59	$5.414 \cdot 10^{-2}$
3-Viterbi	-193.56	$2.213 \cdot 10^{-2}$
4-Viterbi	-193.55	$1.090 \cdot 10^{-2}$

Algoritm	$\ln p(\hat{x}\|y)$	LV
2-Blokk	-194.31	$7.722 \cdot 10^{-1}$
3-Blokk	-193.91	$3.751 \cdot 10^{-1}$
4-Blokk	-193.76	$2.249 \cdot 10^{-1}$
5-Blokk	-193.69	$1.510 \cdot 10^{-1}$

Tabel 13: k-Viterbi ja k-Blokk monotoonsus

k-Viterbi	
Permutatsioon	kogus
0 1 2 3	9316
1 0 2 3	241
0 2 1 3	149
0 1 3 2	71
2 0 1 3	44
1 3 0 2	34
1 2 0 3	30
ülejäänud	115

k-Blokk	
Permutatsioon	kogus
0 1 2 3	9904
1 0 2 3	31
1 2 3 0	14
1 2 3 0	10
0 1 3 2	7
0 3 1 2	7
2 0 1 3	5
ülejäänud	22

Tabel 14: Iteratiivsete algoritmide tulemused

Algoritm	W-Viterbi		0-Viterbi		1-Viterbi	
	$\ln p(\hat{x}\|y)$	LV	$\ln p(\hat{x}\|y)$	LV	$\ln p(\hat{x}\|y)$	LV
esialgne	-194.79	1.256	-197.23	3.693	-193.78	$2.409 \cdot 10^{-1}$
SEM	-194.31	$7.774 \cdot 10^{-1}$	-194.64	1.10	-193.76	$2.190 \cdot 10^{-1}$
HC	-194.78	1.241	-197.04	3.502	-193.77	$2.344 \cdot 10^{-1}$
HCSEM	-194.31	$7.688 \cdot 10^{-1}$	-194.62	1.083	-193.75	$2.134 \cdot 10^{-1}$
SEMHC	-194.31	$7.689 \cdot 10^{-1}$	-194.62	1.083	-193.75	$2.131 \cdot 10^{-1}$

Algoritm	2-Blokk		$\min_{x_t} \|y_t - x_t\|$	
	$\ln p(\hat{x}\|y)$	LV	$\ln p(\hat{x}\|y)$	LV
esialgne	-194.31	$7.722 \cdot 10^{-1}$	-219.93	$2.640 \cdot 10$
SEM	-194.00	$4.658 \cdot 10^{-1}$	-196.52	2.979
HC	-194.30	$7.612 \cdot 10^{-1}$	-212.78	$1.924 \cdot 10$
HCSEM	-193.99	$4.582 \cdot 10^{-1}$	-196.31	2.774
SEMHC	-193.99	$4.575 \cdot 10^{-1}$	-196.31	2.777

Tabel 15: Keskmine iteratsioonide arv

Algoritm	W-Viterbi	0-Viterbi	1-Viterbi	2-Blokk	$\min_x \|y_t - x_t\|$
SEM	0.534	0.765	0.032	0.396	1.197
HC	0.060	0.164	0.014	0.033	2.41

Tabel 16: Iteratiivsete algoritmide tulemused juhuslikel alglähenditel

	$\ln p(\hat{x}\|y)$	LV
alglähendite keskmine	-209.70	20.62
maksmimaalne alglähend	-199.13	10.06
maksimaalne SEM	-189.56	0.48
maksimaalne HC	-197.63	8.56

Kokkuvõte

Antud magistritöö eesmärk oli esitada ja uurida potentsiaalseid lähendusalgoritme Viterbi raja leidmiseks kolmekaupa Markovi mudelitel. Kolmekaupa Markovi ahelad üldistavad varjatud Markovi ahelaid, mida rakendatakse paljudes valdkondades, näiteks kõne tuvastamises ja bioinformaatikas.

Töö esimeses osas on lühidalt tutvustatud Markovi ahelate definitsioone, omadusi ja näiteid. Teises peatükis antakse ülevaade neljast erinevast lähendusalgoritmist: k-Viterbi, k-blokk ning iteratiivsetest algoritmidest segmenteerimis EM-algoritm (SEM) ja mäkke ronimise algoritm (HC). Mainitud algoritmid põhinevad varem olemasolevatel algoritmidel, mis said kohandatud kolmekaupa Markovi mudelitele. Iteratiivsete algoritmide korral näidati, et neid rakendades väljundid ei kehvene.

Kolmandas peatükis võrreldi numbriliste simulatsioonide põhjal lähendusalgoritmide sooritusvõimet ning uuriti nende omadusi. Esiteks võrreldi omavahel k-Viterbi ja k-blokk algoritmi sooritusvõimet ning monotoonsust k kasvades. Veel võrreldi omavahel iteratiivseid meetodeid, segmenteerimis EM (SEM) ja mäkke ronimise (HC) algoritmi, rakendades neid erinevatele alglähenditele. Üldiselt kõige parem sooritusvõime oli k-Viterbi algoritmil. Iteratiivsetest algoritmidest toimis kõige paremini SEM algortim, aga selle sooritusvõime ei olnud märkimisväärne võrreldes k-Viterbi algoritmiga. Saadi teada, et HC ja SEM algoritm ei jaga alati püsipunkte, mis teeb võimalikuks nende algoritmide vaheldumisi kasutamise.

Praktikas, kui on soov leida võimalikult hea lähend Viterbi rajale, on soovitatav kasutada k-Viterbit algoritmi, ajalisest ja arvutimälulisest piirangutest tulenevalt maksimaalse võimaliku k väärtusega, ning soovi korral rakendada sellele veel mõnda iteratiivsetest algoritmidest. Edasist uurimist vajab, kas mingist k väärtusest alates on k-Viterbi väljund Viterbi rada ning kas on võimalik hinnata kui suur peab k olema, et k-Viterbi annaks soovitud täpsusega.

Viited

[Ava21] K. Avans. Paarikaupa markovi mudel: definitsioon ja näited. 2021.

[Cha10] Djalil Chafaï. The dirichlet markov ensemble. *Journal of Multivariate Analysis*, 101(3):555–567, 2010.

[DP04] S. Derrode and W. Pieczynski. Signal and image segmentation using pairwise markov chains. *IEEE Transactions on Signal Processing*, 52(9):2477–2489, 2004.

[FG02] R. Fernandez and A. Galves. Markov approximations of chains of infinite order. *Bulletin Brazilian Mathematical Society*, 33:295–306, 11 2002.

[GGMP18] I. Gorynin, H. Gangloff, E. Monfrini, and W. Pieczynski. Assessing the segmentation performance of pairwise and triplet markov models. *Signal Processing*, 145:183–192, 2018.

[GLT13] S. Gallo, M. Lerasle, and D.Y. Takahashi. Markov approximation of chains of infinite order in the $\bar{d}$-metric. *Markov Processes and Related Fields*, 1, 01 2013.

[KL22] K. Kuljus and J. Lember. Hybrid classifiers of pairwise markov models, 2022.

[Lei14] Leiger, T. and Zolk, I. Matemaatiline analüüs iv, loengukonspekt, 2014.

[Lem22] Lember, J. Informatsiooniteooria, loengukonspekt ja ülesanded, 2022.

[LGKK19] J. Lember, D. Gasbarra, A. Koloydenko, and K. Kuljus. Estimation of viterbi path in bayesian hidden markov models. *METRON*, 77(2):137–169, may 2019.

[LK14] J Lember and A. Koloydenko. Bridging viterbi and posterior decoding: A generalized risk approach to hidden path inference based on hidden markov models. *Journal of Machine Learning Research*, 15(1):1–58, 2014.

[LLLP11] P. Lanchantin, J. Lapuyade-Lahorgue, and W. Pieczynski. Unsupervised segmentation of randomly switching data hidden with non-gaussian correlated noise. *Signal Processing*, 91(2):163–175, 2011.

[Loe04] H.-A. Loeliger. An introduction to factor graphs. *IEEE Signal Processing Magazine*, 21(1):28–41, 2004.

[TG17] T. Tennisberg and K. Gabrel. *Võistlusprogrammeerimine 1. osa*. Tartu Ülikool, 2017.

[Vik23] Vikipeedia. Algoritmide tüübid — vikipeedia, 2023. [Võrgus; vaadatud: 26. juuli 2023].

[Wik23a] Wikipedia contributors. Dirichlet distribution — Wikipedia, the free encyclopedia, 2023. [Online; accessed 11-August-2023].

[Wik23b] Wikipedia contributors. Expectation–maximization algorithm — Wikipedia, the free encyclopedia, 2023. [Online; accessed 12-July-2023].

[Wik23c] Wikipedia contributors. Hill climbing — Wikipedia, the free encyclopedia, 2023. [Online; accessed 16-July-2023].

[Wik23d] Wikipedia contributors. Logsumexp — Wikipedia, the free encyclopedia, 2023. [Online; accessed 27-July-2023].

[Wik23e] Wikipedia contributors. Markov kernel — Wikipedia, the free encyclopedia, 2023. [Online; accessed 20-August-2023].

Lisad

I. Simulatsioonides kasutatud Pythoni kood

Simulatsioonide jaoks kasutatav .ipynb fail on leitav aadressil `https://gitlab.ut.ee/oskar.soop/kolmekaupa-markovi-ahelate-viterbi-raja-laehendamine`.

Sama koodi peamised funktsioonid ja klassid on lisatud ka alljärgnevalt.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from scipy.special import logsumexp
from itertools import product

#The following two functions are modifications of logsumexp and is used in SEM algorithm to calculate sum p log p.
#Inspired by https://github.com/scipy/scipy/blob/v1.11.1/scipy/special/_logsumexp.py#L7-L128
from scipy._lib._util import _asarray_validated
def sumexp(a, axis=None, b=None, keepdims=False):
  a = _asarray_validated(a, check_finite=False)
  if b is not None:
    a, b = np.broadcast_arrays(a, b)
    if np.any(b==0):
      a = a + 0.
      a[b == 0] = -np.inf

  a_max = np.amax(a, axis=axis, keepdims=True)

  if a_max.ndim > 0:
    a_max[~np.isfinite(a_max)] = 0
  elif not np.isfinite(a_max):
    a_max = 0

  if b is not None:
    with np.errstate(invalid='ignore'):
      b = np.asarray(b)
      tmp = np.nan_to_num(b * np.exp(a - a_max), neginf=-np.inf)
  else:
    tmp = np.exp(a - a_max)

  if not keepdims:
    a_max = np.squeeze(a_max, axis=axis)

  s = np.sum(tmp, axis=axis, keepdims=keepdims)

  return np.exp(a_max)*s


def logsumexp2(a, axis=None, b=None, keepdims=False, return_sign=False):
    a = _asarray_validated(a, check_finite=False)
    if b is not None:
        a, b = np.broadcast_arrays(a, b)
        if np.any(b == 0):
            a = a + 0.  # promote to at least float
            a[b == 0] = -np.inf

    a_max = np.amax(a, axis=axis, keepdims=True)
```

```
49
50      if a_max.ndim > 0:
51          a_max[~np.isfinite(a_max)] = 0
52      elif not np.isfinite(a_max):
53          a_max = 0
54
55      if b is not None:
56        with np.errstate(invalid='ignore'):
57          b = np.asarray(b)
58          tmp = np.nan_to_num(b * np.exp(a - a_max), neginf=-np.inf, posinf=np.inf)
59      else:
60          tmp = np.exp(a - a_max)
61
62      # suppress warnings about log of zero
63      with np.errstate(divide='ignore'):
64          s = np.sum(tmp, axis=axis, keepdims=keepdims)
65          if return_sign:
66              sgn = np.sign(s)
67              s *= sgn  # /= makes more sense but we need zero -> zero
68          out = np.log(s)
69
70      if not keepdims:
71          a_max = np.squeeze(a_max, axis=axis)
72      out += a_max
73
74      if return_sign:
75          return out, sgn
76      else:
77          return out


1
2   class MultiMarkovChain:
3       def __init__(self, transition_law, transition_prob, initial_state,
        initial_probability, M_U, M_X):
4           """
5           Initialize the MultiMarkovChain class.
6
7           Parameters:
8           - transition_law: Random function to find Z_k+1 using Z_k.
9           - transition_prob: Function that returns P(Z_k+1|Z_k).
10          - initial_state: Initial state (U_1, X_1, Y_1).
11          - initial_probability: Function to calculate initial probability.
12          - M_U: Number of states of U.
13          - M_X: Number of states of X.
14          """
15          self.initial_state = initial_state
16          self.initial_probability = initial_probability
17          self.num_variables = 3
18          self.transition_law = transition_law
19          self.transition_prob = transition_prob
20          self.M_U = M_U
21          self.M_X = M_X
22
23      def generate(self, size):
24          """
25          Generates a Markov chain of size 'size'.
26          """
27          self.N=size
28          self.chain = np.zeros((size, self.num_variables))
29          self.chain[0] = last_state = self.initial_state
30          for i in range(1, size):
31              last_state=self.transition_law(last_state)
```

```
32              self.chain[i]=last_state
33
34      def get_chain(self):
35          """
36          Returns the generated Markov chain.
37          """
38          return self.chain
39
40      def get_marginal(self, i):
41          """
42          Returns the marginal of generated Markov chain.
43
44          Parameters:
45          - i: Index of the marginal (0 = U, 1 = X, 2 = Y).
46          """
47          return self.chain[:,i]
48
49      def get_probability(self,X):
50          """
51          Calculate the log probability ln P(x,y).
52
53          Parameters:
54          - X: Values of the variable X.
55          """
56          N = self.N
57          Y = self.get_marginal(2)
58          M_U = self.M_U
59
60          A = np.zeros((N,M_U))
61          B = np.zeros((N,M_U))
62
63          # Initialize boundary conditions for A and B
64          for i in range(M_U):
65            A[0,i] = np.log(self.initial_probability(i,X[0],Y[0]))
66            B[-1,i] = np.log(1)
67
68          # Calculate AB using dynamic programming
69          for n in range(1,N):
70            A[n]=logsumexp( self.transition_probs[n-1,:,X[n],:,X[n-1]] + A[n-1][np.
    newaxis,:] , axis=-1)
71            B[-n-1]=logsumexp( self.transition_probs[-n,:,X[-n],:,X[-n-1]]+ B[-n][:,np.
    newaxis]     , axis=0)
72          AB = A+B
73          return logsumexp(AB[0])
74
75
76
77      def forw_back(self,K):
78          """
79          Calculates the forward-backward algorithm up to K-th blocks.
80          """
81          N = self.N
82          M_U = self.M_U
83          M_X = self.M_X
84          Y = self.get_marginal(2)
85          transition_prob = self.transition_prob #P(X_k+1|X_k)
86
87          # Initialize arrays for A and B
88          # For each k the shape of A is (n,m_u,m_x,...,m_u,m_x), where there is m_u,m_x
    k times and m_u,m_x to the left on an another m_u,m_x is the past state related to
     the another m_u,m_x (to the left is past).
89          A = [np.zeros([N-k+1]+[M_U,M_X]*k) for k in range(1,K+1)]
```

```
        B = np.zeros((N, M_U, M_X))
        AB= [np.zeros([N-k+1]+[M_U,M_X]*k) for k in range(1,K+1)]

        # Vectorize log transition probabilities
        transition_probs = np.zeros((N-1,M_U,M_X,M_U,M_X))
        for n in range(N-1):
          for i in product(range(M_U),range(M_X), repeat=2):
            transition_probs[(n,)+i]=np.log(transition_prob((i[0],i[1],Y[n+1]),(i[2],i
    [3],Y[n])))

        self.transition_probs = transition_probs

        # Initialize boundary conditions for A and B
        for i in product(range(M_U), range(M_X)):
          A[0][0][i]= np.log(self.initial_probability(*i,Y[0]))
          B[(-1,)+i]= np.log(1)
        for n in range(1,N):
          A[0][n]=logsumexp( transition_probs[n-1] + A[0][n-1][np.newaxis,np.newaxis
    ,:,:] , axis=(-1,-2))
          B[-n-1]=logsumexp( transition_probs[-n]+ B[-n][:,:,np.newaxis,np.newaxis]
     , axis=(0,1))

        # Calculate A and AB for higher block sizes
        for k in range(2,K+1):
          A[k-1] = transition_probs[k-2:].transpose((0,3,4,1,2))[( slice(None), *((None
    ,None)*(k-2)))] + A[k-2][:-1][...,None,None]

        for k in range(1,K+1):
          AB[k-1] = A[k-1] + B[(slice(k-1,None),*((None,None)*(k-1)),slice(None),slice(
    None))]

        self.AB=AB
        self.A = A
        self.B = B


    def block(self,K):
        """
        Perform the k-block estimation.

        Requires that forw_back() is run once with an equal or higher block size.

        Parameters:
        - K: Block size.
        """
        N=self.N
        M_X = self.M_X


        delta = np.zeros([N-K+2]+[M_X]*(K-1)) #N-K+2 blokki
        argmax = np.zeros([N-K+1]+[M_X]*(K-1))

        # ABX is AB summed over U
        ABX = [logsumexp(self.AB[k-1], axis=tuple(range(1,2*k,2))) for k in range(1,K
    +1)]


        # Calculate delta_0
        sum = ABX[0][0]
        for k in range(1,K-1):
            sum = sum[...,None] + ABX[k][0]
            delta[0] = sum
```

```
146
147         # Calculate delta_1,...,n-k
148         for n in range(1,N-K+2):
149             delta[n] = np.max(  delta[n-1,...,None] + ABX[K-1][n-1], axis=0  )
150             argmax[n-1] = np.argmax(  delta[n-1,...,None] + ABX[K-1][n-1], axis=0  )
151
152         # Calculate delta_n-k+1
153         sum=ABX[0][-1]
154         for k in range(1,K-1):
155             sum = sum[None] + ABX[k][-1]
156
157         # Find the argmax of the last k-1 terms.
158         delta_f = delta[N-K+1] + sum
159         argmax_a = np.unravel_index(delta_f.argmax(), delta_f.shape)
160
161         sol = np.zeros(N)
162         sol[-K+1:]=argmax_a
163
164         # Backtrack to fill in the rest of the path
165         sol[-K] = argmax[-1][tuple(sol[-K+1:].astype(int))]
166         for n in range(2,N-K+2):
167             sol[-K+1-n] = argmax[-n][tuple(sol[-K+2-n:-n+1].astype(int))]
168
169         return sol
170
171
172     def EM_log_free(self,q_0,Q):
173       """
174       Perform EM algorithm using without double log expressions.
175       This function is called by EM function.
176       """
177       transition_probs= self.transition_probs
178       N = self.N
179       M_X = self.M_X
180
181       delta = np.zeros([N,M_X])
182       argmax = np.zeros([N-1,M_X])
183
184       #transition_probs (N-1,M_U,M_X,M_U,M_X)
185       f = sumexp(a=Q[...,None,None], b=transition_probs.transpose((0,3,1,4,2)), axis
      =(1,2)) #ln[- sum p(u_t-1=a, u_t=b, (x^i,y)) * ln p(z_t) ]
186
187       f_0 = sumexp(a=q_0[:,None], b=self.A[0][0], axis=0)
188
189       #A[0][0] is ln p(u_0,x_0,y) of shape (M_U,M_X). q_0 is ln p(u_0,(x^i,y)) of shape
      (M_U)
190       delta[0] = f_0
191
192       for n in range(1,N):
193         delta[n] = np.max(  delta[n-1,:,None] + f[n-1], axis=0  )
194         argmax[n-1] = np.argmax(  delta[n-1,:,None] + f[n-1], axis=0  )
195
196       argmax_a = np.argmax(delta[N-1])
197
198       sol = np.zeros(N)
199       sol[-1] = argmax_a
200
201       sol[-2] = argmax[-1][sol[-1].astype(int)]
202       for n in range(2,N):
203         sol[-1-n] = argmax[-n][tuple(sol[-n:-n+1].astype(int))]
204
205       return sol
```

```
206
207
208
209
210     def EM(self,X_0):
211       """
212       Perform the EM algorithm.
213
214       Parameters:
215       - X_0: Initial values for variable X.
216       """
217       N = self.N
218       M_U = self.M_U
219       M_X = self.M_X
220       Y = self.get_marginal(2)
221       transition_prob = self.transition_prob #P(X_k+1|X_k)
222       transition_probs= self.transition_probs
223       A = np.zeros((N, M_U))
224
225       A2 = np.zeros([N-1,M_U,M_U])  #(to the left is past)
226
227       B = np.zeros((N, M_U))
228
229       Q = np.zeros([N-1,M_U,M_U]) #ie A2B
230
231       transition_probs_U = np.zeros((N-1,M_U,M_U))
232       for n in range(N-1):
233         transition_probs_U[n] = transition_probs[n,:,X_0[n+1],:,X_0[n]]
234
235
236       for i in range(M_U):
237         A[0][i]= np.log(self.initial_probability(i,X_0[0],Y[0]))
238         B[-1,i]= np.log(1)
239       for n in range(1,N):
240         A[n]=logsumexp( transition_probs_U[n-1] + A[n-1][np.newaxis,:] , axis=(-1))
241         B[-n-1]=logsumexp( transition_probs_U[-n]+ B[-n][:,np.newaxis]     , axis=(0))
242
243       A2 = transition_probs_U.transpose((0,2,1)) + A[:-1][...,None]
244
245       q_0 = A[0] + B[0] #ln p(u_0,(x^i,y)) shape (M_U)
246
247       Q = A2 + B[1:,None] # Q=ln p(u_t-1,u_t,(x^i,y))
248
249       delta = np.zeros([N,M_X])
250       argmax = np.zeros([N-1,M_X])
251
252       f = logsumexp2(a=Q[...,None,None], b=-transition_probs.transpose((0,3,1,4,2)),
      axis=(1,2)) #ln[- sum p(u_t-1=a, u_t=b, (x^i,y)) * ln p(z_t) ] NB! Extra log(-...)
      !!!
253
254       f_0 = logsumexp2(a=q_0[:,None], b=-self.A[0][0], axis=0) #NB! Extra log(-...)
255
256       delta[0] = f_0
257
258
259       with np.errstate(invalid='raise'):
260         try:
261          for n in range(1,N):
262           delta[n] = np.min( np.logaddexp(delta[n-1,:,None],f[n-1]), axis=0 )
263           argmax[n-1] = np.argmin( np.logaddexp(delta[n-1,:,None],f[n-1]), axis=0 )
264         except:
265           return self.EM_log_free(q_0,Q)
```

```
    argmax_a = np.argmin(delta[N-1])

    sol = np.zeros(N)
    sol[-1] = argmax_a

    sol[-2] = argmax[-1][sol[-1].astype(int)]
    for n in range(2,N):
      sol[-1-n] = argmax[-n][tuple(sol[-n:-n+1].astype(int))]

    return sol


  def HC(self, X, iterations = 0):
    """
    Perform the Hill Climbing algorithm.

    Parameters:
    - X: Initial state sequence.
    - iterations: Number of iterations performed.

    Returns:
    - Updated state sequence after applying Hill Climbing.
    - Total number of iterations.
    """
    N = self.N
    Y = self.get_marginal(2)
    M_U = self.M_U
    M_X = self.M_X

    A = np.zeros((N,M_U))
    B = np.zeros((N,M_U))

    # Calculate initial A and B values
    for i in range(M_U):
      A[0,i] = np.log(self.initial_probability(i,X[0],Y[0]))
      B[-1,i] = np.log(1)
    for n in range(1,N):
      A[n]=logsumexp( self.transition_probs[n-1,:,X[n],:,X[n-1]] + A[n-1][np.newaxis
,:] , axis=-1)
      B[-n-1]=logsumexp( self.transition_probs[-n,:,X[-n],:,X[-n-1]]+ B[-n][:,np.
newaxis]     , axis=0)
    AB = A+B

    max_p = logsumexp(AB[0])
    A_mod = np.zeros(M_U)
    B_mod = np.zeros(M_U)
    grad = None
    # Iterate over time steps and possible state changes
    for n in range(1,N-2):
      for x in range(M_X):
        A_mod = logsumexp( self.transition_probs[n-1,:,x,:,X[n-1]] + A[n-1][np.
newaxis,:] , axis=-1)
        B_mod = logsumexp( self.transition_probs[n+1,:,X[n+1],:,x]+ B[n+1][:,np.
newaxis] , axis=0)
        prob_mod = logsumexp(A_mod + B_mod)
        if (prob_mod > max_p ):
          max_p = prob_mod
          grad = (n,x)

    # Termination conditions
    if (grad is None or iterations==10*N):
      return X, iterations
```

```
    # Apply state modification
    X_mod = X.copy()
    X_mod[grad[0]] = grad[1]

    # Check if the modification is the same as the original state
    if (np.array_equal(X_mod, X)):
      return X, iterations

    # Recur with modified state
    return self.HC(X_mod , iterations + 1)



  def EM_til_fixed(self,X_0):
    """
    Perform EM iterations until a fixed point is reached.

    Parameters:
    - X_0: Initial state sequence.

    Returns:
    - Estimated state sequence after EM iterations.
    - Number of iterations performed.
    """
    for i in range(100):
      X_1=self.EM(X_0).astype(int)
      if(np.array_equal(X_1,X_0)):
        return X_1, i
      X_0=X_1
    return X_1, i+1



  def k_Viterbi(self,K=2):
    """
    Perform the k-Viterbi algorithm.

    Parameters:
    - K: Block size.
    """
    N=self.N
    M_X = self.M_X


    delta = np.zeros([N-K+2]+[M_X]*(K-1)) #N-K+2 blokki
    argmax = np.zeros([N-K+1]+[M_X]*(K-1))

    #AB[k][n,M_U,M_X,...]
    #ABX is AB summed over U
    #ABX[k][n,M_X,M_X,...]
    ABX = [logsumexp(self.AB[k-1], axis=tuple(range(1,2*k,2))) for k in range(1,K+1)]

    delta[0] = 0 #sum

    # Calculate delta_1...,n-k
    for n in range(1,N-K+1):
      delta[n] = np.max(  delta[n-1,...,None] + ABX[K-1][n-1] - np.nan_to_num(ABX[K
-2][n], neginf=np.inf) , axis=0  )
      argmax[n-1] = np.argmax(  delta[n-1,...,None] + ABX[K-1][n-1] - np.nan_to_num(
ABX[K-2][n], neginf=np.inf), axis=0  )

    # Calculate delta_n-k+1 and argmax_n-k+1
```

```
      delta[N-K+1] = np.max(  delta[N-K,...,None] + ABX[K-1][N-K] , axis=0  )
      argmax[N-K] = np.argmax(  delta[N-K,...,None] + ABX[K-1][N-K], axis=0  )

      #argmax_a is argmax_a( delta_n-k+1(a) )
      argmax_a = np.unravel_index(delta[N-K+1].argmax(), delta[N-K+1].shape)

      sol = np.zeros(N)
      sol[-K+1:]=argmax_a

      sol[-K] = argmax[-1][tuple(sol[-K+1:].astype(int))]
      # Backtrack to fill in the rest of the path
      for n in range(2,N-K+2):
        sol[-K+1-n] = argmax[-n][tuple(sol[-K+2-n:-n+1].astype(int))]

      return sol


    def Viterbi_W(self):
      """
      Perform the W-Viterbi algorithm to find the Viterbi path of PMM (W, Y).
      """
      N=self.N
      M_X = self.M_X
      M_U = self.M_U

      delta = np.zeros((N-1,M_U*M_X))
      argmax = np.zeros((N-2,M_U*M_X))

      delta[0]=0

      # Calculate delta and argmax iteratively
      for n in range(1,N-1):
        delta[n] = np.max( delta[n-1,...,None] + self.AB[1][n-1].reshape((M_U*M_X,M_U*
    M_X)) - np.nan_to_num(self.AB[0][n].reshape(M_U*M_X), neginf=np.inf), axis=0 )
        argmax[n-1] = np.argmax( delta[n-1,...,None] + self.AB[1][n-1].reshape((M_U*M_X
    ,M_U*M_X)) - np.nan_to_num(self.AB[0][n].reshape(M_U*M_X), neginf=np.inf), axis=0
    )

      delta_f = delta[N-2,...,None] + self.AB[1][-1].reshape((M_U*M_X,M_U*M_X))
      argmax_a = np.unravel_index(delta_f.argmax(), (M_U*M_X,M_U*M_X))

      sol = np.zeros(N)
      sol[-2:] = argmax_a

      # Backtrack to fill in the rest of the path
      for n in range(3,N+1):
        sol[-n] = argmax[-n+2][sol[-n+1].astype(int)].astype(int)

      return np.unravel_index(sol.astype(int), (M_U,M_X))


    def sample(self):
      """
      Generates a Markov chain (U,X|Y)
      """
      AB = self.AB
      N = self.N
      M_U = self.M_U
      M_X = self.M_X
      sample = np.zeros((N,2))

      sample[0] = np.random.randint(0,2,2) #depends on chain
```

```
443
444       for i in range(1,N):
445         sample[i] = np.unravel_index(
446             np.random.choice(M_U*M_X,
447                              p=np.exp(
448                                 (AB[1][i-1,sample[i-1,0].astype(int),sample[i-1,1].
      astype(int)] - np.nan_to_num(AB[0][i-1,sample[i-1,0].astype(int),sample[i-1,1].
      astype(int),None,None], neginf=np.inf)).flatten())
449                              ),
450                               (M_U,M_X)
451                     )
452
453       return sample[:,1]
454
455
456     def iterative_sample(self,num_samples=100):
457       """
458       Perform iterative sampling and evaluations.
459
460       Parameters:
461       - num_samples: Number of samples to generate and evaluate.
462
463       Returns:
464       - Average sample probability.
465       - Best sample probability.
466       - Best SEM probability.
467       - Best HC probability.
468       """
469       best_SEM = - np.inf
470       best_HC  = - np.inf
471       best_sample = - np.inf
472       avg_sample = 0
473       for i in range(num_samples):
474         sample = self.sample()
475         sample_prob = self.get_probability(sample.astype(int))
476         avg_sample += sample_prob
477         cur_SEM = self.get_probability(self.EM_til_fixed(sample.astype(int))[0])
478         cur_HC  = self.get_probability(self.HC(sample.astype(int))[0])
479         if(best_SEM < cur_SEM):
480           best_SEM = cur_SEM
481         if(best_HC < cur_HC):
482           best_HC = cur_HC
483         if(best_sample < sample_prob):
484           best_sample = sample_prob
485       avg_sample = avg_sample/num_samples
486       return avg_sample, best_sample, best_SEM, best_HC
487
488 # End of the class MultiMarkovChain.
```

```
1  # Import library for printing tables
2  from tabulate import tabulate
3
4  # Print some results in tabular format
5  def report(MC, k=5):
6    #actual
7    actual = MC.get_chain()[:,1].astype(int)
8
9    #Viterbi on (W,Y)
10   Viterbi_W  = MC.Viterbi_W()[1]
11   Viterbi_W_prob = MC.get_probability(Viterbi_W)
12
13   #PMAP aka 0-Viterbi
```

```
PMAP = np.argmax(logsumexp(MC.AB[0], axis=1),axis=1)
PMAP_prob = MC.get_probability(PMAP)

#k-block & k-Vitebi
k_blocks = np.zeros((k-1,N))
k_blocks_probs = np.zeros(k-1)
k_block_table = []
k_Viterbis = np.zeros((k-1,N))
k_Viterbis_probs = np.zeros(k-1)
k_Viterbi_table = []
for i in range(2,k+1):
  k_blocks[i-2] = MC.block(i)
  k_blocks_probs[i-2] = MC.get_probability(k_blocks[i-2].astype(int))
  k_block_table += [[str(i)+"-block", k_blocks_probs[i-2]]]
  k_Viterbis[i-2] = MC.block_minus_PMAP(i)
  k_Viterbis_probs[i-2] = MC.get_probability(k_Viterbis[i-2].astype(int))
  k_Viterbi_table += [[str(i-1)+"-Viterbi", k_Viterbis_probs[i-2]]]


#monotonicity
print("k-Viterbi mono: ",np.argsort(k_Viterbis_probs))
print("k-block mono: ",np.argsort(k_blocks_probs))



print(tabulate([
    ["actual", MC.get_probability(actual)],
    ["Viterbi_V", Viterbi_V_prob],
    ["0-Viterbi", PMAP_prob],
]+k_Viterbi_table+k_block_table))


print(tabulate([
    ["algorithm","initial","SEM","HC","mixed"],
    ["W-Viterbi",Viterbi_W_prob,MC.get_probability(MC.EM_til_fixed(Viterbi_W)[0]),MC.
  get_probability(MC.HC(Viterbi_W)[0]),MC.EM_HC_mixed(Viterbi_W)],
    ["0-Viterbi",PMAP_prob,MC.get_probability(MC.EM_til_fixed(PMAP)[0]),MC.
  get_probability(MC.HC(PMAP)[0]),MC.EM_HC_mixed(PMAP)],
    ["1-Viterbi",k_Viterbis_probs[0],MC.get_probability(MC.EM_til_fixed(k_Viterbis
  [0].astype(int))[0]),MC.get_probability(MC.HC(k_Viterbis[0].astype(int))[0]),MC.
  EM_HC_mixed(k_Viterbis[0].astype(int))],
    ["2-blokk",k_blocks_probs[0],MC.get_probability(MC.EM_til_fixed(k_blocks[0].
  astype(int))[0]),MC.get_probability(MC.HC(k_blocks[0].astype(int))[0]),MC.
  EM_HC_mixed(k_blocks[0].astype(int))]
],headers="firstrow"))
```

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Oskar Soop,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose ”Kolmekaupa Markovi ahelate Viterbi raja lähendamine“, mille juhendaja on Jüri Lember, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.
2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creati-ve Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.
3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Oskar Soop

19.05.2023