Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Miha Filej

Naslov diplomskega dela

diplomsko delo
univerzitetni študij računalništva in informatike

Mentor: dr. Andrej Brodnik

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Besedilo teme diplomskega dela študent prepiše iz študijskega informacijskega sistema, kamor ga je vnesel mentor. V nekaj stavkih bo opisal, kaj pričakuje od kandidatovega diplomskega dela. Kaj so cilji, kakšne metode uporabiti, morda bo zapisal tudi ključno literaturo.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Miha Filej sem avtor diplomskega dela z naslovom:

Naslov diplomskega dela (angl. *Diploma thesis sample*)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom dr. Andreja Brodnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 15. junija 2016 Podpis avtorja:

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
BSD	Berkeley Software Distribution	Berkley distribucija programske opreme
EM	Expectation–Maximization	maksimizacija pričakovane vrednosti
GPL	GNU General Public License	splošno dovoljenje GNU
HMM	Hidden Markov Model	skriti markovski model (SMM)
LGPL	GNU Lesser General Public License	splošno večje dovoljenje GNU
NLG	Natural Language Generation	tvorjenje naravnega jezika
NLP	Natural Language Processing	procesiranje naravnega jezika
OCR	Optical Character Recognition	optično prepoznavanje znakov
OTP	Open Telecom Platform	odprta platforma za telekome
TEI	Text Encoding Initiative	iniciativa za kodiranje besedila

Povzetek

Naslov: Naslov diplomskega dela

Področje NLG se ukvarja s tvorjenjem naravno zvenečih besedil. Cilj diplomskega dela je ugotoviti, do kolikšne mere lahko kompleksna pravila tvorjenja naravnega jezika posnemamo s statističnimi sistemi, natančneje s skritimi markovskimi modeli. Delo predstavi potrebno teoretično podlago za obstoj skritih markovskih modelov in opiše njihovo uporabo pri tvorjenju besedil. V okviru diplomskega dela je opravljen tudi pregled obstoječih orodij za delo s skriti markovskimi modeli, medsebojna primerjava orodij in pregled njihove primernosti za uporabo pri tvorjenju besedil. Opisan je postopek implementacije knjižnice za delo s skritimi markovskimi modeli v programskem jeziku Elixir. Dve izmed pregledanih orodij in implementirana knjižnica so uporabljeni za tvorjenje besedil na podlagi korpusa slovenskega pisnega jezika. Izbere se kriterij za primerjavo tvorjenih besedil, ki se uporabi za primerjavo modelov, kot tudi za primerjavo tvorjenih besedil s korpusom.

Ključne besede: tvorjenje naravnega jezika, skriti markovski modeli, algoritem Baum-Welch, algoritem Forward-Backward, algoritem EM, Elixir, Erlang/OTP.

Abstract

Title: Diploma thesis sample

Natural language generation (NLG) is the task of producing text that feels natural to the reader. The goal of this diploma thesis is to study to which level natural language generation can be achieved using statistical models – specifically hidden Markov models. The diploma thesis covers probability and information theories that allow the definition of hidden Markov models and describes how such models can be used for the purpose of text generation. Available tools for working with hidden markov models are reviewed, compared, and assesed for their suitability for generating text. A library for hidden Markov models is implemented in Elixir. Two of the reviewed tools and the implemented library are used to generate text from a corpus of written slovenian language. A criterion for comparing generated texts is chosen and used to compare the models as well as comparing the generated texts to the corpus.

Keywords: natural language generation, hidden markov models, Baum-Welch algorithm, Forward-Backward algorithm, expectation–maximization algorithm, Elixir, Erlang/OTP.

Poglavje 1

Uvod

V želji po večji produktivnosti, zmanjševanju obsega dela in višji kakovosti življenja avtomatizacija pronica v vsa področja človeškega življenja. Stroji prevzemajo nove odgovornosti v vseh panogah, med drugim tudi v jezikoslovju.

Obdelava naravnega jezika (NLP) je področje računalništva, jezikoslovja in umetne inteligence, ki se že desetletja ukvarja z reševanjem mnogih izzivov v povezavi z naravnim oz. človeškim jezikom, kot so na primer strojno prevajanje, prepoznavanje in tvorjenje govora, optično prepoznavanje znakov (OCR), preverjanje slovnične pravilnosti, analiza čustvenosti, zlogovanje besed ter tvorjenje besedil.

Tvorjenje naravnega jezika (NLG) je proces sestavljanja besedil, ki zvenijo naravno in bi za njih lahko sklepali, da jih je proizvedel človek [23].

Nastopa lahko v različnih obsegih. Ljudi lahko popolnoma nadomesti, kot na primer pri verbalizaciji vremenske napovedi, ki jo lahko stroji opravijo povsem samostojno. Poveča lahko človeško storilnost, recimo s pisanjem osnutkov dokumentov ali s strnjevanjem besedil v obnove. Ne nazadnje lahko pozitivno vpliva tudi na kakovost življenja (tvorjenje besedil, ki pomagajo lajšati anksioznost; verbalizacija grafov, tabel in drugih nebesednih oblik za slabovidne ljudi). Drugi primeri uporabe NLG sistemov vključujejo obnavljanje in strjevanje zdravniških, inženirskih, finančnih, športnih ... podatkov;

tvorjenje osnutkov dokumentov, kot so navodila za uporabo, pravni dokumenti, poslovna pisma ... [9]

Na slovenskem spletu je bila med leti 2007 in 2015 na voljo Virtualna davčna asistentka VIDA. Ta pogovorni sistem je uporabnikom omogočal, da s pisnim postavljanjem vprašanj v naravnem jeziku dobijo odgovore na splošna in najpogostejša vprašanja iz področja dohodnine [24].

Motivacija za tvorjenje naravnega jezika v okviru našega diplomskega dela je priprava besedil za predmet digitalna forenzika [15]. Gre za zgodbe, ki opisujejo okoliščine namišljenega kaznivega dejanja. Za čim boljšo izkušnjo študentov pri reševanju nalog bi bilo popolno, da bi bila vsaka zgodba unikatna. Ker pa je pisanje zgodb dolgotrajen proces, bi sestavljanje takšnega števila besedil za profesorje in asistente predstavljalo preobsežno količino dela. Če bi imeli NLG sistem, ki bi bil zmožen tvorjenja unikatnih zgodb, bi lahko študentom omogočili bolj kakovostno izkušnjo, hkrati pa olajšali delo profesorjem in asistentom.

NLG sistemi se delijo na statistične sisteme in na sisteme, osnovane na znanju[1]. Slednji so za tvorjenje odvisni od znanja iz neke določene domene. Čeprav so taki sistemi sposobni proizvesti kakovostno besedilo, zahtevajo pri izgradnji veliko truda in sodelovanja s končnimi uporabniki sistema. Poleg tega lahko taki sistemi pokrivajo samo domeno, za katero so bili zgrajeni. Na drugi strani so statistični sistemi enostavnejši za izgradnjo in lažji za prilagoditev na druge domene. Slabost statističnih sistemov je njihova izpostavljenost napakam — zaradi manjšega števila omejitev v postopku tvorjenja lahko proizvedejo tudi besedila brez smisla [23].

Področje tvorjenja naravnega jezika je ena izmed zadnjih panog računalniškega jezikoslovja, ki so privzele statistične metode [29].

V diplomskem delu bomo ugotavljali, ali so skriti markovski modeli lahko primerna statistična metoda za tvorjenje naravnega jezika.

[1] angl. *knowledge-based*

1.1 Struktura naloge

Najprej bomo opravili kratek pregled področij verjetnostne teorije in teorije informacije, ki omogočajo teoretično podlago za skrite markovske modele in predstavili, kako lahko s takimi modeli tvorimo zaporedja. Predstavili bomo vprašanja, ki se pojavijo pri uporabi skritih markovskih modelov v praksi in odgovorili na tista, ki so ključna za izvedbo naše naloge.

Enačbe, pridobljene iz odgovorov, bomo v 3. poglavju preslikali v psevdokodo, ki nam bo omogočila delo s skritimi markovskimi modeli. Opisali bomo, s katerimi težavami smo se srečali pri preslikavi matematičnih enačb in kako smo jih rešili.

V 4. poglavju bomo opravili pregled področja obstoječih orodij za delo s skritimi markovskimi modeli. Opisali bomo, kako smo orodja izbrali, jih pregledali, preverili njihovo ustreznost za našo problemsko domeno, na koncu pa jih na kratko medsebojno primerjali. V 5. poglavju bomo razložili, zakaj smo se odločili za implementacijo lastne knjižnice za delo s skritimi markovskimi modeli. Opisali bomo, kako smo se naloge lotili in kako smo izbrali orodja ter pristope. Navedli bomo poglavitne težave pri implementaciji in njihove rešitve.

V 6. poglavju bomo preverili, kako se skriti markovski modeli obnesejo pri tvorjenju naravnih besedil. Uporabili bomo lastno implementacijo skritih markovskih modelov in dve izmed orodij, pregledanih v 4. poglavju. Izbrali smo korpus slovenskega pisnega jezika, ga pripravili za učenje modelov, zgradili modele in jih uporabili za tvorjenje besedil. Nastala besedila smo primerjali med seboj in z izvornim korpusom. Za merilo primerjave smo izbrali število glagolov na stavek. Preverili smo, ali različna orodja in spreminjanje parametrov pri teh privedejo do različnih rezultatov.

Poglavje 2

Skriti markovski modeli

Za verjetnostne porazdelitve, ki sestavljajo statistične modele, predpostavljamo, da približno ocenjujejo pojave iz realnega sveta. Ti približki so včasih dovolj dobri, da lahko pojave z zadostno natančnostjo napovedujemo ali tvorimo za statistično populacijo reprezentativnega vzorca. Primer takih statističnih modelov so skriti markovski modeli. Uporabni so pri določanju zaporedja nukleotidov molekul DNK, prepoznavanju govora, napovedovanju gibanj na finančnih trgih ipd. V tem poglavju bomo predstavili teoretične temelje, na katerih so nastavljeni skriti markovski modeli.

Verjetnostna teorija je ključnega pomena pri razumevanju, izražanju in obravnavi koncepta negotovosti. Skupaj s teorijo odločanja omogočata, da se na podlagi vseh razpoložljivih informacij, pa čeprav nepopolnih ali dvoumnih, optimalno odločamo [5].

V nadaljevanju bomo po [5] povzeli nekaj konceptov verjetnostne teorije, ki so ključnega pomena za razlago teoretičnih osnov skritih markovskih modelov.

Pravilo vsote:

$$p(X) = \sum_{Y} p(X, Y)$$

Pravilo produkta:

$$p(X, Y) = p(Y|X)p(X)$$

$p(X, Y)$ je v tem primeru presek ali produkt verjetnosti, ki ga opišemo kot verjetnost, da se zgodita X in Y. Na podoben način je količina $p(Y|X)$ pogojna verjetnost oz. verjetnost, da se zgodi Y glede na X.

Bayesovo pravilo (2.1) določa zvezo med pogojnimi verjetnostmi.

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)} \tag{2.1}$$

Spremenljivki X in Y sta *neodvisni*, kadar velja (2.2).

$$p(X, Y) = p(X)p(Y) \tag{2.2}$$

Stohastični proces, včasih imenovan *naključni* proces, je zbirka naključnih spremenljivk, ki predstavljajo spreminjanje nekega sistema skozi čas [56].

2.1 Diskretni viri informacij

Teorija informacij obravnava *diskretne vire informacije*, tj. *naključne procese*, ki oddajajo informacijo, zajeto v diskretnih signalih. Modeliranje diskretnih virov temelji na opazovanju nizov simbolov, ki jih viri oddajajo. Končni, neprazni množici simbolov

$$\check{\mathbf{Z}} = \{v_1, v_2, \ldots, v_K\},\ K \in \mathbb{N}$$

pravimo tudi *abeceda vira*. Simboli, ki jih vir oddaja, ustrezajo nizu naključnih spremenljivk

$$\{X_t,\ t = 1, 2, \ldots, n\}.$$

Označimo jih z $X_1, X_2, \ldots, X_n$, kjer X_n označuje n-ti simbol oddanega zaporedja. Enačba (2.3) opisuje porazdelitev verjetnosti, da vir odda znak x_1 v trenutku $t = 1$, znak x_2 v trenutku $t = 2$, ... in znak x_n v trenutku $t = n$.

Enačba (2.4) definira lastnost *stacionarnosti*. Za stacionarne vire pravimo, da se njihove verjetnostne lastnosti s časom ne spreminjajo [37].

$$P(X_1 = x_1, \ldots, X_n = x_n) \geq 0 \tag{2.3}$$

$$P(X_{k+1} = x_1, \ldots, X_{k+n} = x_n) = P(x_1, \ldots, x_n) \tag{2.4}$$

Lastnost *ergodičnosti* pomeni, da lahko porazdelitev verjetnosti vira[1] določimo na podlagi enega samega, ustrezno dolgega niza simbolov [37].

Diskretne vire delimo na:

- vire *brez spomina*, za katere velja, da verjetnost v danem trenutku oddanega simbola ni odvisna od predhodno oddanega zaporedja simbolov;
- vire *s spominom*, za katere velja, da je oddaja simbola v danem trenutku odvisna od števila (k) predhodno oddanih simbolov. Število k določa *red* vira.

Vire s spominom prvega reda imenujemo *markovski viri*[2] (2.5). Za njih velja, da je oddaja simbola v trenutku t odvisna le od simbola, ki je oddan v trenutku $t - 1$.

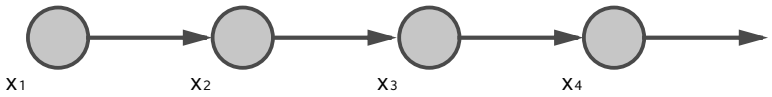
Enačba (2.6) definira še *prehodno verjetnost* p_{ij}, tj. verjetnost, da je vir v trenutku $n + 1$ oddal znak $x_j \in A$ pri pogoju, da je v trenutku n oddal znak $x_i \in A$ [37].

$$\begin{aligned} &P(X_{n+1} = x_{n+1} \mid (X_n = x_n, \ldots, X_1 = x_1)) = \\ =&P(X_{n+1} = x_{n+1} \mid X_n = x_n) \end{aligned} \tag{2.5}$$

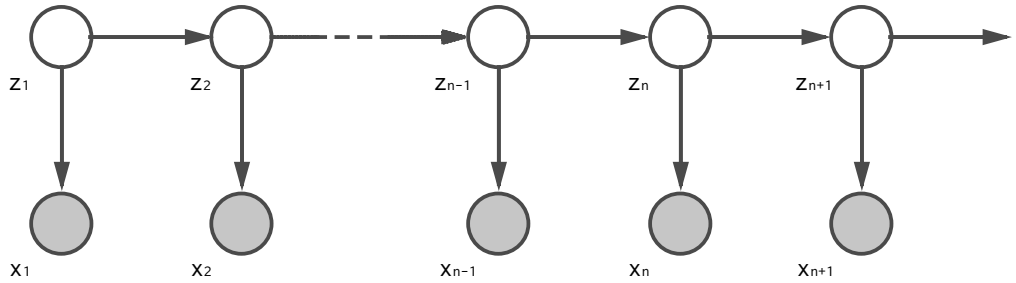
$$p_{ij} = P(X_{n+1} = x_j \mid X_n = xi) \tag{2.6}$$

[1] Tudi jezike lahko definiramo kot ergodične vire. [6]

[2] V slovenski literaturi najdemo prevoda *markovov* [37] in *markovski* [16]. V tem besedilu smo se odločili za uporabo slednjega.



Slika 2.1: Markovska veriga prvega reda.



Slika 2.2: Markovska veriga z latentnimi spremenljivkami.

Če privzamemo, da vsebuje abeceda $|\check{\mathbf{Z}}| = m$ simbolov, potem lahko določimo $1 \leq i, j \leq m$. Vrednosti p_{ij} sestavljajo *matriko prehodnih verjetnosti*:

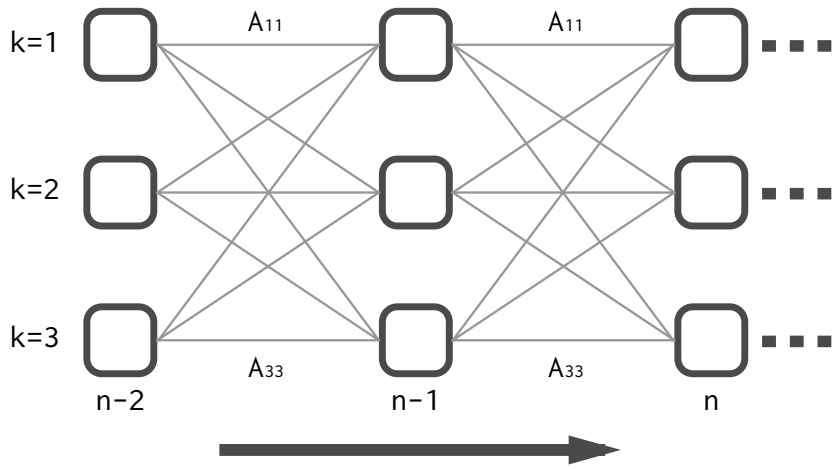
$$P = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{bmatrix}$$

Markovska veriga ali *markovski model* je stohastični proces, za katerega velja markovska lastnost (2.5) [5].

2.2 Skriti markovski modeli

Na sliki 2.1 je markovska veriga prvega reda z množico opazovanj x_n, kjer je porazdelitev $p(x_n|x_{n-1})$ posameznega opazovanja x_n odvisna od izida predhodnega opazovanja x_{n-1}.

Slika 2.2 prikazuje markovsko verigo z latentnimi (neopazovanimi ali pri-



Slika 2.3: Skriti markovski model, predstavljen na razprt način.

kritimi) spremenljivkami $\{z_n\}$. Vsako opazovanje $\{x_n\}$ je pogojeno pripadajoči latentni spremenljivki. Te so vidne v stolpcih diagrama na sliki 2.3, ki prikazuje prehode med stanji v različnih trenutkih. To je osnova, iz katere med drugim izhajajo tudi skriti markovski modeli [5].

Za markovske modele velja, da so v vsakem trenutku v enem izmed N stanj iz množice $Q = \{Q_1, Q_2, \ldots, Q_N\}$. Ob trenutkih $t = 0, 1, \ldots, T$ prehajajo med različnimi stanji $Q_n, n \in N$.

Skriti markovski modeli so izpeljanka markovskih modelov, za katere opazovalci poznajo le verjentostno funkcijo stanja. Stanje, v katerem se model v določenem trenutku nahaja, pa opazovalcem ostane skrito [28].

Za sledeče definicije smo vpeljali označbo (2.7), ki označuje, da se model ob trenutku t nahaja v skritem stanju Q_n.

$$q^{[t]} = Q_n \tag{2.7}$$

Skriti markovski model λ je definiran v obliki[1]

$$\lambda = (A, B, \pi, N, M), \tag{2.8}$$

kjer so A, B, π, N in M parametri, ki opisujejo model [40]:

N ... Število stanj modela.

M ... Število različnih opazovanih simbolov oz. velikost abecede.

A ... Matrika verjetnosti prehodov stanj $A = [a_{ij}]$, ki opisuje verjetnost, da se bo sistem ob trenutku $t + 1$ znašel v stanju Q_j ob dejstvu, da je bil ob trenutku t v stanju Q_i:

$$a_{ij} = P(q^{[t+1]} = Q_j \mid q^{[t]} = Q_i), \qquad 1 \leq i, j \leq N. \tag{2.9}$$

B ... Matrika verjetnosti oddanih simbolov $B = [b_j(k)]$, ki opisuje verjetnost, da bo sistem, ki se nahaja v stanju Q_j, oddal simbol $v_k \in \check{\mathbf{Z}}$:

$$b_j(k) = P(v_k \mid q^{[t]} = Q_j), \qquad \begin{aligned} &1 \leq j \leq N, \\ &1 \leq k \leq M. \end{aligned} \tag{2.10}$$

π ... Začetna porazdelitev stanj $\{\pi_i\}$, kjer velja:

$$\pi_i = P(q^{[0]} = Q_i), \qquad 1 \leq i \leq N. \tag{2.11}$$

Parametri T, K in O pa opisujejo opazovana zaporedja:

K ... število vseh simbolov abecede, $K = |\check{\mathbf{Z}}|$.

T ... dolžina opazovanega zaporedja.

O ... opazovano zaporedje, kjer opazovanja O_t predstavljajo simbole abecede $\check{\mathbf{Z}}$.

$$O = O_1 O_2 \cdots O_T \tag{2.12}$$

[1] Zapis običajno poenostavimo kot $\lambda = (A, B, \pi)$.

2.2.1 Uporaba modela za tvorjenje zaporedij

Primerno definirani skriti markovski modeli lahko tvorijo zaporedja simbolov dolžine T na naslednji način:

1. izberemo začetno stanje $q^{[0]} = Q_i$ glede na π;
2. nastavimo $t = 0$;
3. izberemo izhodni simbol $O_t = v_k$ glede na trenutno stanje Q_i in porazdelitev verjetnosti, ki jih določa $b_i(k)$;
4. opravimo prehod v novo stanje $q^{[t+1]} = Q_j$ glede na prehodne verjetnosti iz stanja Q_i, ki jih določa a_{ij};
5. nastavimo $t = t + 1$; če je $t < T$ se vrnemo na točko 3; sicer postopek zaključimo.

Navedeni postopek lahko uporabimo tako za tvorjenje zaporedij simbolov, kot za ugotavljanje, na kakšen način je določeno opazovano zaporedje najverjetneje nastalo [40].

2.2.2 Temeljni problemi skritih markovskih modelov

Pred uporabo skritih markovskih modelov za reševanje praktičnih izzivov moramo najprej rešiti temeljne probleme, ki smo jih povzeli po [40]:

1. Glede na dano opazovano zaporedje O (2.12) in model λ (2.8) moramo določiti $P(O|\lambda)$ — tj. verjetnost opazovanega zaporedja glede na model. Ta problem lahko zastavimo tudi kot vprašanje: Kako dobro se določen model prilega danemu opazovanemu zaporedju? Če izbiramo med konkurenčnimi modeli, nam odgovor na to vprašanje pomaga izbrati najboljšega.

2. Glede na dano opazovano zaporedje O in model λ moramo izbrati pripadajoče zaporedje stanj $Q = q_1 q_2 \cdots q_T$, ki se najbolje prilega opazovanemu zaporedju (ga najbolj smiselno pojasnjuje). Ta problem obravnava *skriti* del modela.

3. Prilagoditi moramo parametre modela λ, tako da maksimiziramo $P(O|\lambda)$. Opazovana zaporedja, ki jih uporabljamo za optimizacijo parametrov modela, imenujemo *učna zaporedja*, ker z njimi model učimo.

Rešitev zadnjega izmed zgoraj naštetih problemov je običajno najpomembnejša, ker nam omogoča modeliranje pojavov iz resničnega sveta [40].

V poglavjih 2.3 in 2.4 bomo predstavili rešitve prvega in tretjega problema, ki so ključne za izvedbo našega dela.

2.3 Algoritem *Forward-backward*

Forward-backward algoritem zahteva izračun vrednosti α (naprej, angl. *forward*) in β (nazaj, angl. *backward*).

Vrednost *forward*

$$\boldsymbol{\alpha_t(i)} = P(O_1 O_2 \cdots O_t, q^{[t]} = Q_i \mid \lambda)$$

opisuje verjetnost, da se dani model λ po delnem opazovanem zaporedju $O_1 O_2 \cdots O_t$ znajde v stanju Q_i. Začetna enačba

$$\alpha_1(i) = \pi_i b_i(O_1), \qquad 1 \leq i \leq N. \tag{2.13}$$

in induktivni del

$$\alpha_{t+1}(j) = [\sum_{i=1}^{N} \alpha_t(i) a_{ij}] b_j(O_{t+1}), \qquad 1 \leq t \leq T-1, \quad 1 \leq j \leq N. \tag{2.14}$$

nam omogočata izračun te vrednosti.

Vrednost *backward*

$$\boldsymbol{\beta_t(i)} = P(O_{t+1}O_{t+2}\cdots O_T \mid q^{[t]} = Q_i, \lambda)$$

opisuje verjetnost delnega opazovanega zaporedja $O_tO_{t+1}\cdots O_T$, ob pogoju, da je bil model λ ob trenutku t v stanju Q_i. Izračun poteka na podoben način, le da se najprej postavi končna vrednost 1:

$$\beta_T(i) = 1, \qquad 1 \leq i \leq N, \tag{2.15}$$

nato pa se izračunajo ostale vrednosti v nasprotnem vrstnem redu od $T-1$ do 1:

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij}b_j(O_{t+1})\beta_{t+1}(j), \qquad 1 \leq j \leq N,$$
$$t = T-1, T-2, \ldots, 1. \tag{2.16}$$

Ko izračunamo vrednost α, dobimo rešitev prvega problema, opisanega v poglavju 2.2.2. S pomočjo vrednosti α lahko izrazimo tudi oceno primernosti modela $P(O \mid \lambda)$.

$$P(O \mid \lambda) = \sum_{i=1}^{N} \alpha_T(i) \tag{2.17}$$

2.4 Algoritem *Baum-Welch*

Za opis algoritma *Baum-Welch* je potrebno najprej definirati še vrednosti ξ in γ.

$$\begin{aligned}\boldsymbol{\xi_t(i,j)} =& P(q_t = S_i, q_{t+1} = S_j \mid O, \lambda) \\ =& \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}\end{aligned} \tag{2.18}$$

opisuje verjetnost, da je glede na dano zaporedje O model λ ob trenutku t v stanju Q_i in v stanju Q_j v trenutku $t+1$.

$$\begin{aligned}\boldsymbol{\gamma_t(i)} &= P(q_t = S_i \mid O, \lambda) \\ &= \sum_{j=1}^{N} \xi_t(i,j) \end{aligned} \tag{2.19}$$

opisuje verjetnost, da se model λ pri opazovanju zaporedja O ob trenutku t znajde v stanju Q_i.

S pomočjo ξ in γ lahko izvedemo ponovno oceno parametrov modela in tako dobimo nov model $\bar{\lambda} = (\bar{A}, \bar{B}, \bar{\pi})$. Pričakovano število prehodov iz katerega koli stanja v stanje Q_i opišemo z

$$\sum_{t=1}^{T=1} \gamma_t(i) \ ,$$

pričakovano število prehodov iz stanja Q_i v stanje Q_j pa z

$$\sum_{t=1}^{T=1} \xi_t(i,j) \ .$$

Ocena verjetnosti, da ob začetnem trenutku $t = 1$ izbrano stanje Q_i:

$$\bar{\boldsymbol{\pi}}_i \ = \ \gamma_1(i) \ . \tag{2.20}$$

Nove verjetnosti prehodov stanj $\bar{a}_{ij}$ ocenimo z razmerjem med pričakovano frekvenco prehodov iz stanja Q_i v stanje Q_j in pričakovanim številom prehodov iz katerega koli stanja v stanje Q_i:

$$\bar{\boldsymbol{a}}_{ij} \ = \ \frac{\sum\limits_{t=1}^{T=1} \xi_t(i,j)}{\sum\limits_{t=1}^{T=1} \gamma_t(i)} \ . \tag{2.21}$$

Nove verjetnosti za oddajo simbolov $\bar{b}_j(k)$ ocenimo na podlagi razmerja med pričakovanim številom pojavov simbola v_k v stanju Q_j in skupni frekvenci

stanja Q_j:

$$\bar{\boldsymbol{b}}_j(k) \;=\; \frac{\sum\limits_{\substack{t=1 \\ O_t=v_k}}^{T=1} \gamma_t(j)}{\sum\limits_{t=1}^{T=1} \gamma_t(j)} \,. \tag{2.22}$$

V literaturi [4], [26], [42] je dokazano, da za tako pridobljen model $\bar{\lambda}$ velja ena izmed naslednjih dveh točk:

1. $\boldsymbol{P(O \mid \bar{\lambda}) > P(O \mid \lambda)}$, kar pomeni, da smo dobili nov model $\bar{\lambda}$, ki bolje pojasnjuje zaporedje O kot njegov predhodnik;
2. $\boldsymbol{P(O \mid \bar{\lambda}) = P(O \mid \lambda)}$, torej $\boldsymbol{\bar{\lambda} = \lambda}$.

Na podlagi zgornjih dveh točk lahko model iterativno izboljšujemo tako, da λ vsakič zamenjamo z izračunanim $\bar{\lambda}$ (1. točka), dokler ne najdemo novega modela (2. točka) [40].

Poglavje 3

Modeliranje skritih markovskih modelov

Eden izmed večjih izzivov našega diplomskega dela je bila preslikava matematičnih algoritmov za modeliranje skritih markovskih modelov v programsko kodo. V sledečem poglavju bomo opisali postopek preslikave. Za ključne algoritme bomo navedli psevdokodo, na koncu pa bomo razložili, s kakšnimi težavami smo se srečevali in na kakšen način smo jih rešili.

Izgradnjo modela smo zastavili na način, prikazan na sliki 3.1. Postopek vključuje glavno zanko, ki izvaja Baum-Welchev algoritem (glej poglavje 2.4). Ko se dovolj približamo kritični točki, se zanka prekine.

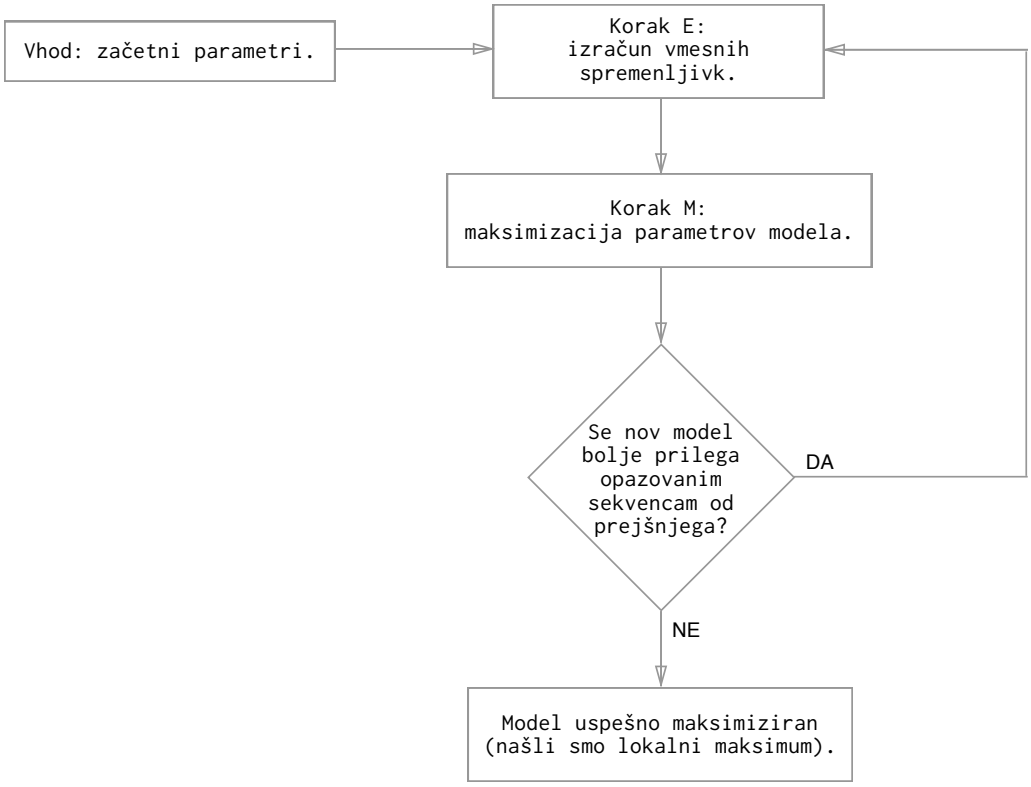
Algoritem EM je sestavljen iz dveh korakov – koraka E in koraka M. V koraku E izračunamo vmesne vrednosti α, β, γ in ξ, s pomočjo katerih ocenimo trenutno verjetnost modela

$$\lambda = (a, b, \pi) \ .$$

V koraku M na podlagi dobljenih vrednosti izračunamo nov model

$$\bar{\lambda} = (\bar{a}, \bar{b}, \bar{\pi}) \ .$$

V nadaljevanju predstavljamo psevdokodo za izračun vrednosti α, β, γ in ξ.



Slika 3.1: Potek postopne izgradnje modela.

3.1 Korak E

V koraku E poteka izračun vrednosti spremenljivk α, β, γ in ξ ter izračun verjetnosti modela v trenutni iteraciji.

Izračun spremenljivke α poteka v fukciji `estimate_alpha` (3.1), ki je preslikava enačb (2.13) in (2.14).

Algoritem 3.1 `estimate_alpha`

for $i \leftarrow 1$ to N **do**
  $\alpha_1(i) = \pi_i \cdot b_i(O_1)$
for $t \leftarrow 2$ to T **do**
  for $j \leftarrow 1$ to N **do**
    $sum = 0$
    for $i \leftarrow 1$ to N **do**
      $sum = sum + \alpha_{t-1}(i) \cdot a_{ij}$
    $\alpha_t(j) = \sum \cdot b_j(O_t)$

Podobnost definicij α in β se pokaže tudi v simetriji njunih algoritmov. Algoritem `estimate_beta` (3.2) ima drugačen izraz za izračun vsote v notranji zanki. Glavna zanka se v tem primeru izvaja od zadnjega stanja proti prvemu. Algoritem je preslikava enačb (2.15) in (2.16).

Algoritem 3.2 `estimate_beta`

for $i \leftarrow 1$ to N **do**
  $\beta(i) = 1$
for $t \leftarrow T - 1$ to 1 **do**
  for $j \leftarrow 1$ to N **do**
    $sum = 0$
    for $i \leftarrow 1$ to N **do**
      $sum = sum + \beta_{t+1}(j) \cdot a_{ij} \cdot b_j(O_{t+i})$
    $\beta_t(i) = sum$

Ko pridobimo vrednosti α in β, lahko nadaljujemo z izračunom verjetnosti

modela za posamezno stanje glede na dano opazovano zaporedje. Na podlagi definicije (2.18) vrednost ξ preslikamo v funkcijo `estimate_xi` (3.3).

Algoritem 3.3 `estimate_xi`

for $t \leftarrow 1$ to $T-1$ **do**
  for $i \leftarrow 1$ to N **do**
    for $j \leftarrow 1$ to N **do**
      $\xi_t(i,j) = \alpha_t i \cdot a_{ij} \cdot b_j(O_{t+1}) \cdot \beta_{t+1}(j)$

Na podlagi enačbe (2.19) lahko vrednost γ na poenostavljen način izrazimo v funkciji `estimate_gamma` (3.4).

Algoritem 3.4 `estimate_gamma`

for $t \leftarrow 1$ to $T-1$ **do**
  for $i \leftarrow 1$ to N **do**
    $sum = 0$
    for $j \leftarrow 1$ to N **do**
      $sum = sum + \xi_t(i,j)$
    $\gamma_t(i) = sum$

S pomočjo spremenljivke α lahko, kot je pokazano v (2.17), izračunamo tudi verjetnost modela glede na dano opazovano sekvenco `compute_model_probability` (3.5).

Algoritem 3.5 `compute_model_probability`

$prob = 0$
for $i \leftarrow 1$ to N **do**
  $prob = prob + \alpha_T(i)$
return $ln(prob)$

`ln`$(\cdot)$ predstavlja funkcijo naravnega logaritma. V tem delu se bo beseda *logaritem* nanašala izključno na naravni logaritem.

3.2 Korak M

Cilj koraka M je na podlagi izračunanih vrednosti γ in ξ ponovno oceniti parametre $\bar{\pi}$, $\bar{a}$ in $\bar{b}$ ter tako pridobiti nov model $\bar{\lambda}$.

Funkcija `reestimate_pi` (3.6) verjetnosti začetnih stanj $\bar{\pi}$ izračuna tako, da enostavno prebere izračunane vrednosti spremenljivke $\bar{\gamma}$ za prvi simbol opazovane sekvence, kot to določa enačba (2.20).

Algoritem 3.6 `reestimate_pi`

```
1: for i ← 1 to N do
2:     π_i = γ_1 i
```

Sledita še preslikava enačbe za izračun nove vrednosti $\bar{a}$ (2.21) v funkcijo `reestimate_a` (3.7) in enačbe za izračun nove vrednosti $\bar{b}$ (2.22) v funkcijo `reestimate_b` (3.8).

Algoritem 3.7 `reestimate_a`

```
1: for i ← 1 to N do
2:     for j ← 1 to N do
3:         numerator = 0
4:         denominator = 0
5:         for t ← 1 to T − 1 do
6:             numerator = numerator + ξ_t(i, j)
7:             denominator = denominator + γ_t(i)
8:         ā_ij = numerator/denominator
```

3.3 Iterativna maksimizacija parametrov

Predhodno smo definirali vse ključne funkcije za maksimizacijo paramterov modela, sledi pa še povezava v glavno zanko, prikazano na sliki 3.1. V literaturi [54] najdemo dokaze, da maksimizacija modela nujno vodi k povečanju verjetnosti modela do stopnje, ko ta konvergira proti kritični točki. Naš

Algoritem 3.8 `reestimate_b`

O_t je simbol na mestu t v opazovani sekvenci, v_k pa k-ti simbol iz abecede.

1: **for** $i \leftarrow 1$ to N **do**
2: $numerator = 0$
3: $denominator = 0$
4: **for** $t \leftarrow 1$ to $T, k \leftarrow 1$ to K **do**
5: $denominator = denominator + \gamma_t(i)$
6: **if** $O_t = v_k$ **then**
7: $numerator = numerator + \gamma_t(i)$
8: $\bar{b}_i(k) = numerator/denominator$

program lahko torej zasnujemo tako, da iteracijo nadaljuje do omenjene kritične točke oz. njenega približka, tj. točke, kjer se verjetnosti prejšnjega in trenutnega modela razlikujeta za manj kot izbrano vrednost ε[1]. Da bi se zaščitili pred izbiro premajhne vrednosti ε, glavno zanko še dodatno omejimo z navzgor omejenim maksimalnim številom ponovitev[2], kot to prikazuje algoritem 3.9. Predpostavljamo, da je `model` nek začetni model, vrednosti ε in `max_iter` pa so določili uporabniki.

Algoritem 3.9 Glavna zanka.

```
model_prob = compute_model_probability(model)
loop
    new_model = reestimate_model(model)
    new_model_prob = compute_model_probability(new_model)
    if new_model_prob − model_prob > ε AND max_iter > 0 then
        model_prob = new_model_prob
        model = new_model
    else
        break
```

[1]Izbire vrednosti ε prepustimo uporabnikom, ker različni problemi zahtevajo različne vrednosti. Za relativno enostaven model smo začeli z vrednostjo 10^{-6}.

[2]Tudi ta vrednost je nastavljiva; privzeta omejitev je 100 ponovitev.

3.4 Izbira začetnih parametrov

Postopek iterativnega izboljševanja parametrov modela, opisan v poglavju 3.3, zahteva izbiro začetnih parametrov, ki služijo kot vhod v iteracijo (glej algoritem 3.9). Ustrezna izbira parametra vpliva na to, ali bo maksimizacija privedla samo do lokalnega ali pa do globalnega maksimuma [40]. V literaturi [4], [40] zasledimo dva nezahtevna pristopa, ki se izkažeta za enako dobra: naključne vrednosti in enakomerno[1] [28] razporejene vrednosti (pod pogojem, da se držimo omejitev stohastičnosti in da so verjetnosti neničelne). Izjema je parameter b, za katerega se izkaže, da je ustrezna začetna ocena vrednosti pomembna [40]. Oceno lahko pridobimo na več načinov, odvisno od tipa podatkov. V našem primeru je vhod besedilo, tako da smo za vrednosti vzeli relativne pogostosti pojavljanja simbolov abecede.

3.5 Podpora za mnogotera opazovana zaporedja

Baum-Welchev algoritem je v osnovi definiran za maksimizacijo verjetnosti modela glede na dano opazovano sekvenco. Ker želimo algoritem uporabiti za namen tvorjenja besedila, moramo za uspešno učenje modela uporabiti dovolj veliko učno množico, npr. besedilo s približno 10.000 besedami. Takšno učno zaporedje prinaša dve težavi:

- Opazovano zaporedje takšne dolžine bo v koraku E Baum-Welchevega algoritma za bolj oddaljene simbole abecede izračunalo zelo majhne verjetnosti, ki bodo povzročile napako podkoračitve[2] (podrobnejši opis sledi v razdelku 3.6).

- Zapis celotnega besedila v obliki enega opazovanja sporoča odvisnost zaporedja, kar pomeni, da so povedi, ki nastopijo kasneje, odvisne od predhodnih. Takšna odvisnost je v našem modelu nezaželena (bolj

[1] Prehodu v vsako stanje dodelimo enako verjetnost.
[2] angl. *underflow*

kot odvisnost med povedmi je za nas zanimiva odvisnost med besedami). [56]

V literaturi zasledimo različne pristope k obravnavi mnogoterih zaporedij. Pri enostavnejših pristopih ima vsako zaporedje enako težo [4], [40], pri kompleksnejših pa imajo lahko modeli različne uteži ali pa je njihova izbira celo dinamično prepuščena drugim algoritmom. [26], [56] Zaradi narave našega problema smo se odločili, da bomo vsako poved obravnavali kot neodvisno zaporedje. Algoritme za priredbo Baum-Welchev algoritma za mnogotera zaporedja smo prevzeli iz enačb v članku [41].

Algoritmi (3.1), (3.2), (3.3), (3.4) za izračun vrednosti $\alpha, \beta, \gamma, \xi$ ostanejo nespremenjeni, korak E pa se spremeni do te mere, da vrednosti α, β, γ in ξ računamo za vsako opazovano zaporedje posebej. Če je $\boldsymbol{O}^{(s)}$ s-to zaporedje (oz. s-ti stavek), potem moramo izračunati vrednosti $\alpha^s, \beta^s, \gamma^s$ in ξ^s.

Z dobljenimi vrednostmi najprej izračunamo verjetnosti posameznih zaporedij glede na trenutni model $P(\boldsymbol{O}^{(s)}|\lambda)$, kar storimo z algoritmom 3.5. Skupna verjetnost opazovanih zaporedij $\boldsymbol{O}$ je enaka zmnožku verjetnosti za posamezna zaporedja $\boldsymbol{O}^{(k)}$ [40]:

$$P(\boldsymbol{O}|\lambda) = \prod_{s=1}^{S} P(\boldsymbol{O}^{(k)}|\lambda) \; . \tag{3.1}$$

Vrednosti, izračunane po Algoritmu 3.5, so logaritmi verjetnosti, zato jih lahko enostavno seštejemo.

Sledijo še posodobitve za korak M, za katerega smo priredili algoritme za izračun vrednosti $\bar{\pi}, \bar{a}$ in $\bar{b}$. (3.6) (3.7) (3.8) Algoritmom smo dodali zanko, ki zajema vsa opazovana zaporedja ($k \leftarrow 1$ do S). Vrednosti α, β, γ in ξ smo zamenjali z njihovimi izpeljankami $\alpha^s, \beta^s, \gamma^s$ in ξ^s.

Za preverjanje pravilnosti postopka je pomembno, da lahko za primer posameznega opazovanega zaporedja dobimo enake vrednosti tako s prilagojenim kot z izvirnim algoritmom. Specifikacije za posamezna opazovana zaporedja ohranimo kar se da nedotaknjene.

3.6 Preprečevanje napake podkoračitve

Uporaba skritih markovskih modelov na dolga opazovana zaporedja zahteva računanje z izredno majhnimi verjetnostmi. Te privedejo do nestabilnosti pri izračunavanju števil v plavajoči vejici [30], med njimi tudi do napake podkoračitve.

Podkoračitev se pojavi že po nekaj iteracijah Baum-Welchevega algoritma (spremenljivke dobijo vrednost 0). Razlog je v tem, da ima pri veliki množici vseh možnih zaporedij besed, neko poljubno opazovano zaporedje zelo majhno pogojno verjetnost. Za spopadanje s to težavo obstajata dve najpogostejši rešitvi:

- *lestvičenje*[1] pogojnih verjetnosti na podlagi skrbno izbranih faktorjev;
- zamenjava pogojnih verjetnosti z vrednostmi njihovih logaritmov.

Prednost slednje je v tem, da lahko algoritme spreminjamo postopoma in pravilnost sprememb vseskozi preverjamo. Preverjanje izvajamo tako, da v naših specifikacijah pričakovane vrednosti zamenjamo z njihovimi logaritmi [30].

Vse potrebne priredbe algoritmov v koraku E so zelo nazorno prikazane v članku [30], zato jih tukaj ne bomo posebej navajali.

V koraku M algoritmov ni potrebno spreminjati. Upoštevati je potrebno le, da po zgoraj navedenih spremembah algoritmi vračajo logaritme vrednosti. Zato na vrednosti π_i v 2. koraku algoritma 3.6, vrednosti $\bar{a}_{ij}$ v 8. koraku algoritma 3.7 in vrednosti $\bar{b}_i(k)$ v 8. koraku algoritma 3.8 uporabimo naravno eksponentno funkcijo e^x.

3.7 Simulacija skritih markovskih modelov

Po uspešni maksimizaciji modela lahko pričnemo s simuliranjem oddajanja simbolov. Postopek je definiran v poglavju 2.2.1.

[1] angl. *scaling, rescaling*

Predpostavljamo, da imamo na voljo model $\lambda = (\pi, a, b)$, s pomočjo katerega želimo simulirati oddajanje simbolov abecede. Želena dolžina sekvence je `target_length`. Na voljo imamo naslednje funkcije:

- `push(list, x)`, ki na konec seznama `list` doda element `x`;
- `pick_state(·)` in `pick_symbol(·)`, ki na podlagi razporeditve verjetnosti in trenutnega stanja izbereta novo stanje oz. simbol.

Algoritem 3.10 postopek predstavi v obliki psevdokode.

Algoritem 3.10 Simulacija modela λ.

```
seqeunce = []
state = pick_state(π)
while length(seqeunce) < target_length do
    seqeunce = push(seqeunce, pick_symbol(b, state))
    state = pick_state(a, state)
return sequence
```

Poglavje 4

Pregled področja

Ena izmed težav pri učenju predmeta digitalna forenzika je priprava nalog za študente. Te se študentom dodelijo v obliki zgodbe, ki opisuje kriminalno dejanje, in slike diska, ki predstavlja evidenco primera. Goljufanje (prepisovanje) znotraj večjih skupin študentov pri reševanju nalog bi lahko preprečili tako, da bi vsakemu študentu dodelili individualizirano nalogo. Da bi lahko dosegli avtomatizirano tvorjenje novih nalog, bi potrebovali dovolj velik korpus, da bi na njegovi podlagi lahko izvedli učenje NLG sistema. Nabor podatkov takšne velikosti namreč ni prosto na voljo.

Motivacija našega diplomskega dela je rešiti problem tvorjenja variacij zgodb na podlagi omejenega števila vhodnih besedil. Kot dokaz koncepta smo za problemsko domeno izbrali tvorjenje kratkih povedi oz. stavkov.

V naslednjem poglavju bomo opravili pregled obstoječih orodij za delo s skritimi markovskimi modeli. Najprej bomo opisali, kakšne vrste funkcionalosti pričakujemo od orodja, ki ga želimo uporabiti v problemski domeni tvorjenja stavkov. Nato bomo opisali, kako smo izbrali nekaj najobetavnejših projektov, jih pregledali, primerjali vrste funkcionalnosti, ki jih ti ponujajo, in za vsakega posebej navedli njegovo ustreznost za uporabo v naši problemski domeni. Navedli bomo še nekaj drugih obetavnih projektov, ki ne ustrezajo vsem kriterijem in jih zato tudi nismo podrobneje pregledali.

4.1 Kvalitativni kriteriji

Markovske modele ločimo glede na vrsto vrednosti, ki jih opazujejo oz. oddajajo. Diskretni modeli so tisti, ki oddajajo iz diskretne zaloge vrednosti, zvezni pa tisti, ki oddajajo iz zvezne zaloge vrednosti. Za našo problemsko domeno želimo, da model oddaja diskretne vrednosti, ki jim pravimo tudi simboli. V našem diplomskem delu se bomo zato osredotočili izključno na projekte, ki modelirajo diskretne modele.

V poglavju 3.5 smo razložili potrebo po podpori za mnogotera opazovana zaporedja. To bomo zahtevali tudi pri pregledovanju projektov.

Za vsak projekt smo pregledali tudi ustreznost dokumentacije. Ker gre za knjižnice in uporabniške vmesnike z ukazno vrstico, način uporabe ni očiten. Vmesniki za pravilno rabo zahtevajo določeno mero dokumentacije. Pri nekaterih projektih smo si lahko pomagali s t. i. `README` datotekami, pri drugih pa s primeri uporabe, ki so jih avtorji priročno vključili poleg izvorne kode. V določenih primerih nam je bila v pomoč tudi sama izvorna koda. Kriterij pri preverjanju projektov je bila naša uspešnost pri njihovi uporabi za namen učenja na podlagi korpusa in tvorjenja stavkov (gledali smo na zmožnost tvorjenja, ne na kakovost nastalih besedil).

Sonnenburg [47] je ugotovil, da bi bilo področje programske opreme za strojno učenje bogatejše, če bi projekti poleg izvorne kode vsebovali tudi krajši članek z opisom uporabe.

Na zadnje smo pregledali tudi licenco, pod katero so projekti izdani. Za uspešno uporabo potrebujemo licenco, ki nam bo dovoljevala prosto uporabo.

Povzetek kvalitativnih kriterijev pri pregledu projektov:

- podpora za diskretne skrite markovske modele;
- podpora za mnogotera opazovana zaporedja;
- dokumentacija, ki nam omogoča praktično uporabo orodja;
- licenca, ki nam omogoča prosto uporabo orodja.

4.2 Zbiranje projektov

Projekte smo iskali na spletnem portalu za kolaborativni razvoj programske opreme GitHub (`https://github.com`) in portalih za iskanje znanstvenih člankov Google Scholar (`scholar.google.com`), CiteSeerX (`citeseerx.ist.psu.edu`), arXiv (`arxiv.org`) ter Microsoft Academic Search (`academic.research.microsoft.com`).

V nadaljevanju bomo podrobneje opisali najdene projekte, ki so bili glede na naše kriterije najbolj ustrezni.

4.3 Projekt hmmlearn

URL naslov	Licenca	Jezik
`http://hmmlearn.readthedocs.io`	BSD	Python

Hmmlearn [53] je skupina algoritmov za nenadzorovano[1] učenje skritih markovskih modelov. Orodje je napisano v programskem jeziku Python, programski vmesnik pa je oblikovan po vzoru scikit-learn[2] [38] modula. Združljivost njunih programskih vmesnikov skupaj z dejstvom, da je scikit-learn zelo razširjen projekt, pomeni, da lahko projekt hmmlearn postane zelo zanimiv za široko skupino uporabnikov. Projekt uporabnikom, preko mehanizma dedovanja, nudi podporo za implementacijo verjetnostnih modelov po meri. Podrobnosti so opisane v dokumentaciji projekta.

Ob času pregleda je bila na voljo prva javna različica projekta 0.1.1, izdana februarja 2015. Prihajajoča[3] različica 0.2.0 prinaša veliko novosti,

[1] angl. *unsupervised*

[2] Scikit-learn je modul za programski jezik Python, ki vključuje široko paleto sodobnih algoritmov za nadzorovano in nenadzorovano strojno učenje pri srednje velikih problemih. Modul se osredotoča na enostavnost uporabe, zmogljivost, dokumentacijo in razumljiv programski vmesnik [39].

[3] Različica 0.2.0 je izšla marca 2016.

med drugim tudi sposobnost za učenje na mnogoterih opazovanih zaporedjih. Funkcionalnost je bila sicer že na voljo v t. i. razvojni različici, vendar smo tukaj naleteli na odstopanja med novimi razvojnimi vmesniki in dokumentacijo, ki je bila takrat na voljo samo za prejšnjo različico. To je razlog, da s to knjižnico učenja modelov na mnogoterih zaporedjih v praksi nismo uspeli izvesti.

Kljub temu da je hmmlearn še v zgodnji razvojni fazi, si od tega projekta, zaradi enostavnih programskih vmesnikov, scikit-learn kompatibilnosti in že sedaj obsežne dokumentacije, v prihodnosti veliko obetamo. Upamo, da ga bomo lahko v prihodnosti še preizkusili.

Projekt je izdan pod neomejujočo odprtokodno licenco BSD, kar naredi projekt primeren za rabo v vseh okoljih, vključno z uporabo v komercialne namene [11].

4.4 Projekt UMDHMM

URL naslov	Licenca	Jezik
`http://www.kanungo.com/software/software.html`	GNU GPL	C

UMDHMM Hidden Markov Model Toolkit [21] je projekt Univerze v Marylandu, ki implementira algoritme za delo s skritimi markovskimi modeli Forward-Backward, Viterbi in Baum-Welch.

Za razliko od ostalih projektov, ki smo jih pregledali, UMDHMM ne ponuja funkcij v obliki knjižnice, ki bi jih lahko uporabniki klicali iz lastne programske kode. Glavni vmesnik za uporabo omenjenih algoritmov so programi, namenjeni uporabi preko ukazne vrstice. Projekt obsega programe `genseq`, `testvit`, `esthmm` in `testfor`.

Program `esthmm` je jedro projekta, ki na podlagi podanega zaporedja simbolov z uporabo algoritma Baum-Welch (glej poglavje 2.4), naredi oceno parametrov za skriti markovski model.

Program `genseq` uporabi parametre modela, ki jih je določil program `esthmm`, in na njihovi podlagi tvori naključno zaporedje simbolov.

Program `testvit` z uporabo algoritma Viterbi oceni, katero zaporedje stanj je najbolj verjetno za neko dano zaporedje simbolov.

Program `testfor` z uporabo algoritma Forward (glej poglavje 2.3) izračuna $P(O \mid \lambda)$ — verjetnost opazovanega zaporedja glede na model.

Za primer (prikazan na sliki 4.1) vzemimo uporabo programa za ocenjevanje parametrov modela. Podati je potrebno parametra N in M, ki določata število stanj modela in število simbolov, ki jih model oddaja. Zaporedje, ki predstavlja učno množico za model, je zapisano v datoteki `input.seq` v obliki zaporedja številk stanj. Rezultat bo izpisan na standardni izhod v obliki parametrov modela $\lambda = (A, B, \pi, N, M)$.

Preprostost vmesnika z ukazno vrstico nam je omogočila, da smo na podlagi vhodnega zaporedja na enostaven način zgradili model in pričeli z simulacijo. Slabost tega vmesnika je pomanjkanje podrobnega nadzora nad vnosom vhodnih podatkov, ki bi ga npr. omogočala knjižnica z zbirko funkcij. Program kot vhod sprejme eno zaporedje. Uporabnik nima nadzora nad tem, kako se ta pred učenjem modela razdeli. V projektu smo sicer našli navedbe, da program podpira učenje modela na podlagi mnogoterih opazovanih zaporedij, vendar zaradi omenjenega pomanjkanja nadzora nismo uspeli ugotoviti, po kakšnem načelu se zaporedja obravnavajo.

V literaturi [57] je navedeno, da se projekt uporablja za napovedovanje nastajanja in analizo struktur beljakovinskih molekul. Strokovnjaki navajajo, da so program UMDHMM delno prilagodili. Kljub temu, da licenca GPL, pod katero je projekt izdan, to zahteva, spremenjena izvorna koda ni na voljo [11].

Za branje dokumentacije nas projekt napoti na priloženo datoteko PDF, ki pa vsebuje samo splošne informacije o teoriji skritih markovskih modelov, ne poda pa navodil za uporabo priloženih programov. Vsakega izmed programov lahko v ukazni vrstici poženemo brez parametrov in tako dobimo kratek

```
$ cat input.seq
T= 10
1 1 1 1 2 1 2 2 2 2
$ esthmm -N 3 -M 2 -v input.seq
M= 2
N= 3
A:
0.998560 0.001067 0.002373
0.001582 0.646206 0.354212
0.364515 0.001455 0.636030
B:
0.088618 0.912382
0.999334 0.001666
0.650811 0.350189
pi:
0.001000 0.999800 0.001199
```

Slika 4.1: Primer uporabe orodja `esthmm`.

priročnik uporabe, npr:

```
$ esthmm
Usage1: esthmm [-v] -N <num_states> -M <num_symbols> <file.seq>
Usage2: esthmm [-v] -S <seed> -N <num_states> -M <num_symbols>
<file.seq>
Usage3: esthmm [-v] -I <mod.hmm> <file.seq>
  N - number of states
  M - number of symbols
  S - seed for random number genrator
  I - mod.hmm is a file with the initial model parameters
  file.seq - file containing the obs. seqence
  v - prints out number of iterations and log prob
```

4.5 Projekt GHMM

URL naslov	Licenca	Jeziki
`http://ghmm.sourceforge.net`	GNU LGPL	Python, C

Projekt GHMM je prosto dostopna knjižnica za jezik C, ki vsebuje izvedbo učinkovitih podatkovnih struktur in algoritmov za osnovne ali razširjene skrite markovske modele z diskretnim in zveznim oddajanjem. Projekt vključuje tudi programsko ovojnico za programski jezik Python, ki ponuja prijaznejši vmesnik in nekaj dodatnih funkcionalnosti. V projektu najdemo še grafični urejevalnik modelov, imenovan HMMEd [22].

Knjižnica se uporablja za širok spekter raziskovalnih, akademskih in industrijskih projektov. Področja uporabe vključujejo finančno matematiko (analiza likvidnosti), fiziologijo (analiza EEG podatkov), računsko lingvistiko in astronomijo (klasifikacija zvezd). V literaturi [46] lahko zasledimo, da je projekt GHMM znatno pripomogel k načenjanju nekaterih novih raziskovalnih vprašanj.

Glede na razširjenost uporabe smo pričakovali, da bo dokumentacija za uporabo orodja GHMM obširnejša. Tudi na spletni strani projekta je navedeno, da ima orodje veliko težavo s pomanjkanjem dokumentacije. Potencialne uporabnike namesto tega napotuje k branju Rabinerjevega članka A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition [40], ki je sicer odličen vir za razumevanje skritih markovskih modelov, vendar ne nudi pomoči pri uporabi knjižnice.

S pomočjo komentarjev v programski kodi projekta smo, kljub pomanjkanju dokumentacije, uspeli vzpostaviti enostaven model, nismo pa uspeli pridobiti natančnejšega nadzora nad postopkom učenja, da bi se izognili situacijam, ko postopek maksimizacije ostane v lokalnem maksimumu. Orodje sicer podpira tehnike, kot je npr. vrivanje šuma[1], vendar samo za zvezno, ne pa tudi za diskretno oddajanje, ki ga potrebujemo v našem primeru.

Projekt je izdan pod delno omejujočo licenco LGPL [10], kar bi lahko predstavljalo težavo pri vključevanju v industrijska okolja, predvsem v primerih, ko je za uporabo potrebna sprememba izvorne kode [11].

4.6 Projekt HMM

URL naslov	Licenca	Jezik
`https://github.com/guyz/HMM`	ni podana	Python

Projekt HMM je ogrodje za delo z skritimi markovskimi modeli, zgrajeno na osnovi sklopa programske opreme NumPy[2]. Implementacija algoritmov, tako kot pri mnogih drugih projektih, temelji na Rabinerjevem članku [40]. Projekt vključuje tako diskretne kot zvezne modele. Ena izmed prednosti ogrodja, ki ni bila prisotna pri vseh projektih, je eksplicitna podpora za

[1] angl. *noise injection*

[2] NumPy je okrajšava za Numerical Python. Ta sklop programske opreme je namenjen znanstvenim izračunom v programskem okolju Python in služi kot temelj številnim znanstvenim knjižnicam [51].

```
words = # vsebuje vhodno besedilo (učno množico)
alphabet = Alphabet(words)
hmm_factory = HMMFromMatricesFactory()
hmm = hmm_factory(alphabet, dist, A, B, pi)
seq = EmissionSequence(alphabet, words)
hmm.baumWelch(seq)

hmm.sample(20, 15) # Tvorjenje 20 zaporedij, od
                   # katerih ima vsako 15 simbolov.
```

Slika 4.2: Primer uporabe ogrodja HMM za učenje na podlagi vhodnega besedila.

razširitve, ki uporabnikom omogoča, da napišejo svoje vrste verjetnostnih modelov. Razširitev je omogočena z uporabo dedovanja, podrobnosti pa so opisane v izvorni kodi projekta v datoteki `GMHMM.py`. Poleg celotnih, po meri izvedenih verjetnostnih modelov, se lahko odločimo tudi za delno prilagoditev s prepisom funkcij, ki opazovanim simbolom nelinearno določajo težo (teža je privzeto enaka za vsak opazovan simbol).

Ogrodje HMM je odvisno izključno od omenjenega programskega paketa NumPy. Iz tega razloga se je ogrodje izkazalo enostavnejše za namestitev od večine ostalih projektov.

Zaradi popolnega pomanjkanja dokumentacije smo za zgled uporabe morali pobrskati po izvorni kodi. Izkazalo se je, da ima projekt v datoteki `HMM/hmm/examples/tests.py` nekaj dovolj nazornih primerov uporabe. Tudi programski vmesnik je dovolj intuitiven, da smo lahko vzpostavili učenje skritih markovskih modelov na podlagi besedila. Izvedli smo tudi simulacijo modela. Izvorna koda je dovolj jasno napisana in smiselno organizirana, da smo lahko potrebne podrobnosti implementacije poiskali sami.

Slika 4.2 prikazuje primer uporabe ogrodja za učenje na podlagi vhodnega besedila (polje `words`) in začetnih parametrov modela (večrazsežnih NumPy polj `A` in `B` ter polja `pi`).

Poglavitna ovira pri morebitni uporabi ogrodja HMM je pomanjkanje odprtokodne licence. Programske opreme, ki licence ne vključuje, ne moremo uporabljati, spreminjati ali deliti, če to ni eksplicitno navedeno s strani avtorjev [32]. Menimo tudi, da bi ustrezna, permisivna odprtokodna licenca, k projektu privabila več razvijalcev in tako pripomogla k njegovi višji kakovosti in širši uporabnosti [49].

4.7 Projekt mhsmm

URL naslov	Licenca	Jezik
`https://cran.r-project.org/web/packages/mhsmm`	GNU GPL	R

Mhsmm je sklop programske opreme, zgrajen za sistem za statistično računanje R. Motivacija za nastanek projekta so bile raziskave povezanosti med uspešnostjo razmnoževanja živine in različnih indikatorjev prisotnosti škodljivcev v kmetijstvu. Skriti markovski modeli so bili uporabljeni za dopolnjevanje pomanjkljivih podatkov in združevanje ločenih opazovanih zaporedij, ki so bila vzorčena z različnimi frekvencami. Poglavitne funkcije programskega sklopa so hkratno opazovanje različnih statističnih spremenljivk in podpora za zaporedja z manjkajočimi vrednostmi. Ključni deli programa so napisani v programskem jeziku C, kar zagotavlja njegovo hitrejše izvajanje [33].

Orodje mhsmm lahko modelira tudi t. i. skrite pol-markovske modele. Pri klasičnih skritih markovskih modelih je čas postanka v posameznem stanju geometrično porazdeljen (enakomerni intervali $t, t+1, \ldots$; glej poglavje 2.1). Pri modeliranju marsikaterih realnih problemov je ta porazdelitev nepraktična [33]. Za našo problemsko domeno abstrakcija pol-markovskih modelov ni potrebna. Naša problemska domena zahteva, da model oddaja diskretne vrednosti in je sposoben opazovanja mnogoterih zaporedij.

Orodje uporabnikom omogoča, da poleg vrste vključenih porazdelitev implementirajo tudi porazdelitve po meri. Dodatne porazdelitve se implementirajo s pomočjo uporabniških razširitev. Zaradi neizkušenosti v programskem okolju R te funkcionalnosti nismo preizkusili.

Sposobnost, ki je pri preostalih projektih nismo zasledili, je prikaz tvorjenih vrednosti pri simulaciji modela na zelo nazoren, grafičen način. Primer izpisa je prikazan na sliki 4.3. Horizontalna črta z barvami prikazuje prehajanje med stanji, krivulja pa tvorjene vrednosti.

Dokumentacijo projekta najdemo v obliki datoteke PDF [33], ki nazorno prikaže nekaj primerov uporabe programskega sklopa. Pri njegovem preizkušanju smo ugotovili, da bi projekt potreboval celovitejšo dokumentacijo programskega vmesnika, ki bi razložila uporabo tudi za funkcije, ki v primerih uporabe niso omenjene.

Projekt je izdan pod restriktivno licenco GPL, ki lahko predstavlja oviro pri vključevanju projekta v industrijska okolja, predvsem v primerih uporabe, kjer bi bila potrebne sprememba izvorne kode [11].

4.8 Primerjava

V nadaljevanju bomo opisane projekte primerjali na podlagi kriterijev, določenih v uvodu poglavja (tabela 4.1).

Ugotovili smo, da vsi pregledani projekti podpirajo skrite markovske modele z oddajanjem diskretnih vrednosti. Enako velja tudi za zvezno oddajanje vrednosti, z izjemo projekta `UMDHMM`. Vsi projekti imajo tudi podporo za mnogotera opazovana zaporedja, čeprav v času pregleda različica knjižnice `hmmlearn` še ni bila pripravljena za uporabo. Projekta `mhsmm` in `GHMM` ponujata tudi grafični prikaz markovskih modelov. Slednji dodatno ponuja grafični vmesnik za gradnjo in urejanje modelov.

Projekta `hmmlearn` in `HMM` sta izvedena izključno v programskem jeziku Python, projekt `GHMM` pa ponuja programsko ovojnico za integracijo z jedrom, izvedenim v programskem jeziku C. Priljubljenost jezika Python na področju

	`hmmlearn`	`UMDHMM`	`GHMM`	`HMM`	`mshmm`
diskretni SMM	da	da	da	da	da
zvezni SMM	da	ne	da	da	da
mn. op. zap.[a]	ne[b]	da[c]	da	da	da
razširljivost	da	ne	ne	da	delna
knjižnjica	da	ne	da	da	da
dokumentacija	da	ne	skopa	ne	skopa
licenca	BSD	GPL	LGPL	ni podana	GPL
jezik	Python	C	C, Python	Python	R

[a] mnogotera opazovana zaporedja

[b] Funkcija je sedaj že na voljo v novi različici knjižnjice.

[c] Podpora za mnogotera opazovana zaporedja je omenjena v izvorni kodi, vendar uporabnik nima nadzora nad načinom, kako se vhod, ki je podan kot en niz, deli na več zaporedij.

Tabela 4.1: Primerjava funkcionalnosti in drugih lastnosti projektov.

skritih markovskih modelov gre najverjetneje pripisati splošni priljubljenosti v znanstvenih in akademskih okoljih ter razširjenosti in zrelosti knjižnic za numerično in znanstveno računanje NumPy in SciPy [51].

Z izjemo `UMDHMM` so vsi ostali projekti izvedeni kot knjižnice, ki jih lahko kličemo iz lastne programske kode (v primerih, ko licence to dovoljujejo). Čeprav se nekateri projekti ne opredeljujejo kot knjižnice, ampak kot ogrodja oz. sklopi programske opreme[1], to na naše preverjanje ni imelo bistvenega vpliva. Poleg zmožnosti uporabe v lastni programski kodi smo si ogledali tudi odprtost za razširitve. V tem pogledu sta najbolj odprti knjižnici `HMM` in `hmmlearn`, ki izrecno podpirata možnost implementacije verjetnostnih modelov po meri. Delno razširljivost nudi tudi projekt `mhsmm`, ki omogoča razširitve s porazdelitvami verjetnosti oddajanja po meri.

Dokumentacija projekta `hmmlearn` je po naših ugotovitvah najbolj informativna. Pri nekaterih projektih smo sicer pomanjkanje dokumentacije lahko

[1] angl. *package*

nadoknadili s primeri uporabe in deloma tudi z branjem izvorne kode, vendar pa bi konkretnejša dokumentacija zagotovila večjo uporabnost projektov. S tem bi projekti postali privlačnejši tudi za širšo množico uporabnikov [47].

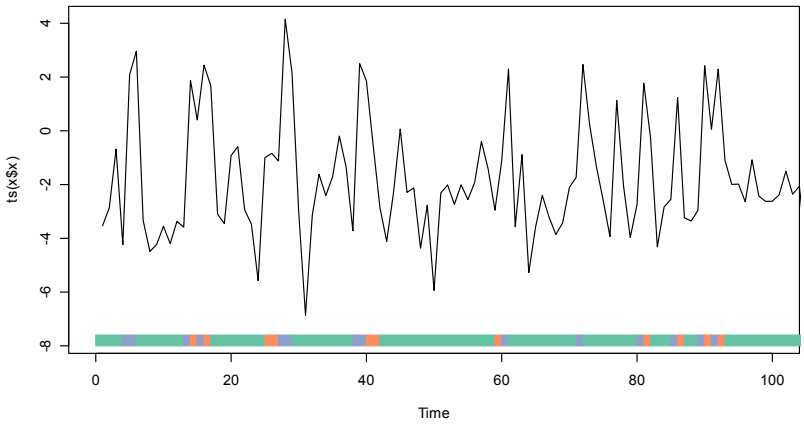
`HMM` je edini izmed pregledanih projektov, ki nima licence. To pomeni, da ga brez avtorjevega dovoljenja ne smemo uporabljati, spreminjati ali deliti. Vsi ostali imajo odprtokodne licence, ki dovoljujejo uporabo projekta vsaj v odprtokodnih okoljih. Najbolj permisiven je `hmmlearn` z BSD licenco, ki predpisuje zelo malo omejitev in dovoljuje vključitev v zaprtokodne in profitne projekte [49].

4.9 Ostali projekti

Omenili bomo še nekaj projektov, ki niso bili ustrezni za primerjavo, vendar za njih menimo, da so vredni omembe.

Zasledili smo nekaj raziskav, ki modeliranje skritih markovskih modelov opravljajo na platformi Matlab/Octave, vendar izvorne kode pri večini ni bilo na voljo [55]. Izjema sta bila projekta iHMM in H2M, ki sta opisana v literaturi. [7], [52] H2M se med drugim uporablja na področjih razpoznavanja govora [43] in na področju zaznavanja ter klasifikacije zvokov (razbitje stekla, človeški vzkliki, streli orožja, eksplozije, zapiranje vrat ...) [12]. Oba projekta sta se za našo problemsko domeno izkazala kot neprimerna, saj podpirata samo zvezne markovske modele, naša domena pa zahteva uporabo diskretnih [7].

Poleg projekta `mhsmm` (glej poglavje 4.7) sta za programsko okolje R na voljo še projekta `HMM` (`https://cran.r-project.org/web/packages/HMM`) in `hsmm` (`https://cran.r-project.org/web/packages/hsmm`), ki se od `mhsmm` razlikujeta v dveh ključnih vidikih [33]. Prvi je ta, da `hsmm` ne podpira uporabniških razširitev za implementacijo novih porazdelitev emisij. Drugi vidik, ki je za našo problemsko domeno pomembnejši, pa je pomanjkanje zmožnosti za obdelavo mnogoterih opazovanih zaporedij, zaradi česa je projekt neprimeren za uporabo pri tvorjenju besedil.



Slika 4.3: Grafični prikaz simulacije skritega markovskega modela z orodjem `mhsmm`.

Poglavje 5

Implementacija knjižnice

Po pregledu obstoječih rešitev v 4. poglavju smo se odločili, da implementiramo svojo knjižnico za delo s skritimi markovskimi modeli. Glavni razlog za to odločitev je bilo splošno pomanjkanje dokumentacije za delo z obstoječimi orodji. Predvidevamo, da nam bo proces implementacije pomagal tudi pri razumevanju problemske domene. Implementacija je na voljo na URL naslovu `https://github.com/mfilej/himamo`.

Problemska domena knjižnice, ki jo želimo implementirati, je učenje skritih markovskih modelov na podlagi daljših besedil. Za učenje modelov želimo uporabiti besedilo ali množico besedil nekega avtorja, ki je dovolj dolga, da predstavlja dobro reprezentacijo pogostosti pojavljanja izrazov oz. besednih zvez in hkrati vsebuje tudi zadosten besedni zaklad. Odločili smo se, da bomo kot učne množice za ta orodja uporabili krajše knjige ali zbirke krajših vrst besedil (esejev, poezije ...). Posamezna učna množica bo tako obsegala od 10.000 do 50.000 besed. Če predpostavimo, da bo vsaka beseda predstavljala en simbol oz. en člen opazovanega zaporedja, potem bi opazovano zaporedje iz take učne množice predstavljalo 10.000 do 50.000 simbolov. Takšna dolžina zaporedja predstavlja težavo za markovske modele, saj začnejo pri izračunu t. i. *forward* in *backward* spremenljivk (glej poglavje 2.3) verjetnosti opazovanih simbolov strmo padati, kar privede do napake podkoračitve (glej poglavje 3.6) [40]. Tej težavi se izognemo tako, da besedilo razdelimo

na več delov (npr. povedi) in vsakega od teh delov obravnavamo kot krajše opazovano zaporedje, neodvisno od ostalih (podrobnosti v poglavju 3.5).

Eden izmed temeljnih ciljev implementacije rešitve je bilo pravilno delovanje programa in z njim pridobljenih rezultatov. Hitrost izvajanja je bila sekundarnega pomena, dokler je bila v praktičnih mejah in ni znatno upočasnjevala razvoja.

5.1 Izbira programskega jezika

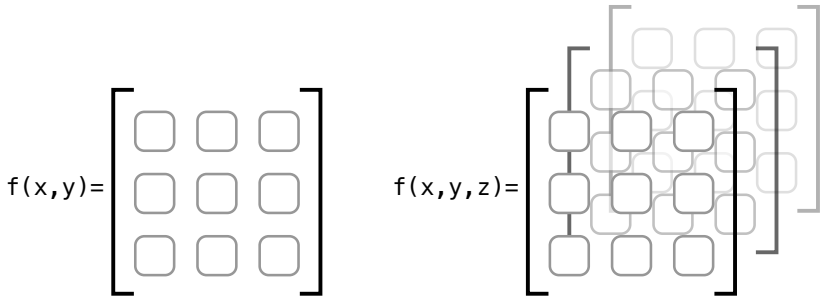
Elixir [50] je sodoben, dinamičen, funkcijski programski jezik. Zgrajen je na osnovi Erlangovega navideznega stroja, zato ima kljub svoji relativni novosti na voljo zelo bogat ekosistem in več desetletij razvoja, ki jih ponuja platforma Erlang/OTP [1]. Ne nazadnje pa izbira programskega jezika za to nalogo predstavlja predvsem subjektivno odločitev avtorjev diplomskega dela.

Da bi si zagotovili čim boljše možnosti za pravilno izvedbo rešitve, smo se odločili za naslednje pristope: uporabo funkcijskega programiranja, premišljeno zasnovane podatkovne strukture (prilagojene za lažje razumevanje) in testiranje enot[1] s sklicevanjem na referenčno rešitev in statično analizo programa. V nadaljevanju bomo podrobneje predstavili vsakega od naštetih pristopov.

5.2 Funkcijsko programiranje in podatkovne strukture

Evolucija funkcijskega programiranja se je začela z delom na lambda računu, nadaljevala s programskimi jeziki, kot so Lisp, Iswim, FM, ML, nato pa s sodobnejšimi različicami, kot so Miranda in Haskell. Za razred *funkcijskih* programskih jezikov je značilno, da izračunavanje izvajajo izključno preko vrednotenja *izrazov* [17].

[1] angl. *unit testing*



Slika 5.1: Dvorazsežna in trorazsežna matrika.

Eden izmed načinov, s katerim funkcijski programski jeziki lajšajo razumevanje programske kode, je odsotnost konstruktov, kot so spremenljivost podatkovnih struktur in odsotnost globalnega stanja. Povzamemo lahko, da se funkcijski programi izogibajo *stranskim učinkom*, ki so pogost vzrok programskih hroščev [19].

Za funkcijske jezike je značilen tudi poudarek na oblikovanju podatkovnih struktur [17]. Nekateri jeziki nudijo tudi t. i. *pattern matching*, ki podajanje podatkovnih struktur znatno olajša [20]. Te značilnosti so se izkazale za zelo uporabne pri implementaciji našega programa.

Najpogostejša oblika podatkov pri implementaciji knjižnice so bili (vmesni ali končni) rezultati funkcij, ki obsegajo dve ali tri razsežnosti. Za njihovo predstavitev smo potrebovali večrazsežnostno podatkovno strukturo (slika 5.1).

Take podatkovne strukture se običajno predstavljajo s pomočjo večrazsežnih polj[1] ali drugih oblik seznamov. V našem primeru se je to izkazalo za nepraktično zaradi podatkovnih tipov, ki so na voljo v programskih okoljih Erlang/OTP in Elixir. Na voljo so seznami povezanega tipa[2], kar pomeni, da niso primerni za indeksirano dostopanje do elementov (`list[x]`). N-terke[3], ki to funkcionalnost podpirajo, pa ne nudijo obširnih funkcionalno-

[1] angl. *multi-dimensional array*
[2] angl. *linked list*
[3] angl. *tuples*

$$f(x,y) = \begin{bmatrix} 0,2 & 0,8 \\ 0,5 & 0,5 \end{bmatrix}$$

```
map = %{
          {0, 0} => 0.2, {0, 1} => 0.8,
          {1, 0} => 0.5, {1, 1} => 0.5,
      }
```

Slika 5.2: Dvorazsežna matrika.

sti za naštevanje[1] [50], ki so pri implementaciji našega programa prav tako nepogrešljive.

Težavi smo odpravili z uporabo podatkovne strukture `map` [31]. `map` je slovar[2], ki v osnovi ponuja preslikavo iz ključa v vrednost[3]. Zaradi narave okolja Erlang/OTP se `map` redno uporablja in je zato njegova implementacija zelo učinkovita. V našem primeru smo `map` uporabili tako, da smo indekse rezultata združili v par in ga uporabili za ključ, sam element rezultata pa zapisali kot vrednost. Slika 5.2 prikazuje primerjavo matematičnega zapisa dvorazsežne matrike in njenega zapisa s podatkovno strukturo `map`.

Za branje in zapisovanje posameznega elementa na danem položaju je bilo dovolj, da smo pravilno sestavili indeksa:

```
iex> map = Map.new
...> map = Map.put(map, {1, 2}, 0.5)
...> Map.get(map, {1, 2})
0.5
```

Standardna knjižnica prav tako podpira naštevanje ali sprehod skozi vse

[1] angl. *enumerate*
[2] angl. *dictionary*
[3] angl. *key-value*

elemente:

```
iex> map = %{...}
...> Enum.map(map, fn({key, value}) ->
...>   # tukaj imamo na voljo spremenljivko ‘element‘
...>   # za poljubno transformacijo
...> end)
```

Da bi zagotovili doslednost pri uporabi opisanih podatkovnih struktur, smo funkcije za branje in zapisovanje vključili v zato namenjen modul.

5.3 Testiranje in preverjanje pravilnosti

Da bi se zavarovali pred lastnimi napakami pri programiranju rešitve, smo pred implementacijo vsake funkcije zapisali njeno specifikacijo v obliki testa enote[1] [8]. Ker je specifikacija izvršljiv program, smo lahko ustreznost napisane kode samodejno preverjali. Testi se izvajajo dovolj hitro, da smo jih lahko med pisanjem vsake funkcije izvršili večkrat in tako preverili, kdaj smo se bližali uspešni rešitvi. Ko je napisana koda ustrezala želenim specifikacijam, smo lahko dodali nove, strožje specifikacije, ali pa nadaljevali s pisanjem naslednje funkcije [2].

Takšni testi so nepogrešljivi tudi pri preurejanju[2], saj moramo biti pri spreminjanju kode pazljivi, da ne pride do nazadovanja[3].

Naša rešitev večinoma obsega matematične algoritme, za katere ni bilo trivialno napisati specifikacije, saj bi lahko tudi z ročnim izračunavanjem hitro uvedli napake. Zato smo se oprli na referenčno implementacijo in s pomočjo pridobljenih rezultatov napisali specifikacije za naše funkcije. Referenčna implementacija, po kateri smo se zgledovali, je Python projekt HMM [58],

[1] angl. *unit test*

[2] Proces, pri katerem spreminjamo interno organiziranost programsko kode, ne da bi pri tem spremenili njeno zunanje vedenje [25].

[3] angl. *regression*

```
assert_in_delta frequencies.a, 2000, 150
assert_in_delta frequencies.b, 8000, 150
```

Slika 5.3: Primer specifikacije s funkcijo `assert_in_delta`.

katerega preglednost in modularnost izvorne kode dovoljujeta posamično izvajanje notranjih funkcij. Pridobljene vrednosti smo prepisali v naše specifikacije.

Zaradi nenatančne narave števil s plavajočo vejico, ki smo jih uporabili za predstavitev večine rezultatov algoritmov, je bilo pri testiranju občasno potrebno prepustiti možnost odstopanja. Določeni deli specifikacije so zato dopuščali, da so dobljene vrednosti od referenčnega rezultata odstopale za izbrano vrednost Δ. To smo dosegli z uporabo ExUnit [14] funkcije `assert_in_delta` (to funkcijo smo nadgradili v `assert_all_in_delta` tako, da lahko preverja več rezultatov naenkrat). Ugotovili smo, da je za večino specifikacij zadostovala vrednost $\Delta = 5^{-8}$.

S pomočjo funkcije `assert_all_in_delta` smo preverili tudi lastnost ergodičnosti funkcije za prehode med stanji z določenimi verjetnostmi. Za primer lahko navedemo model, ki iz določenega stanja dovoljuje prehoda v stanje A z verjetnostjo 0.2 in v stanje B z verjetnostjo 0.8.

Slika 5.3 prikazuje primer specifikacije, ki zahteva, da je od 10.000 poskusov prehoda iz začetnega stanja 2.000 ± 150 takih, kjer se je zgodil prehod v stanje A, 8.000 ± 150 pa takih, kjer se je zgodil prehod v stanje B.

V primerih, ko so bile specifikacije preobsežne, da bi jih ročno zapisali, smo si pomagali z orodji za preverjanje lastnosti [1]. Ta mogočajo, da posamezno lastnost istočasno preverimo na množici števil (npr. na vseh naravnih številih, vseh pozitivnih realnih številih ...). Program nato v procesu preverjanja sam izbere nekaj kombinacij takih števil, za katere preveri ujemanje s pogoji. Za okolje Erlang/OTP je na voljo orodje QuickCheck, katerega smo lahko s pomočjo orodja ExCheck [36] uporabili v programskem okolju Elixir.

[1] angl. *property-based testing*

```
property :ext_log do
  for_all x in such_that(x in real when x > 0) do
    Logzero.ext_log(x) == :math.log(x)
  end
end
```

Slika 5.4: ExCheck specifikacija.

QuickCheck in ExCheck ponujata omejen jezik, s katerim opišemo lastnosti, ki smo jih želeli preveriti v našem programu. Orodji sta generirali naključne testne primere in preverili ustreznost naših rezultatov. [25]

Slika 5.4 prikazuje ExCheck specifikacijo, ki preverja ujemanje vrednosti razširjenega logaritma z vrednostmi običajne logaritemske funkcije za vsa pozitivna realna števila.

5.4 Statična analiza

Tako kot programsko okolje Erlang/OTP, tudi Elixir temelji na konceptu dinamičnih programskih tipov (t.i. *dynamic typing*), ki ne zahtevajo označevanja z informacijami o programskih tipih. Da bi lahko uporabniki kljub temu koristili prednosti, ki jih prinašajo orodja za statično analizo programske kode, lahko uporabijo koncept postopnega tipiziranja (t. i. *gradual typing*) [34]. Na ta način se informacije o programskih tipih dodajajo po potrebi. Orodje za statično analizo na podlagi teh informacij javlja morebitne težave. Rezultat je čistejša programska koda, ki jo je lažje razumeti in vzdrževati. Je robustnejša in bolje dokumentirana [44]. Za statično analizo naše implementacije smo uporabili orodje Dialyxir [18].

5.5 Licenca

Knjižnico smo izdali pod licenco MIT. Licenca je navedena tudi v izvorni kodi v datoteki `LICENCE`.

Poglavje 6

Ovrednotenje

Sistemi za tvorjenje naravnih besedil se običajno ocenjujejo kvantitativno, tj. z merjenjem vpliva besedila na opravljanje določenega opravila (št. potrebnih popravkov besedila; hitrost branja), z anketiranjem ljudi z lestvicami stališč likertovega tipa ali samodejnim izračunavanjem podobnosti tvorjenih besedil s korpusom. Prva dva načina lahko podata dobro oceno NLG sistema, vendar zahtevata veliko človeškega truda in časa. Samodejno izračunavanje lahko po drugi strani oceno ponudi hitro in učinkovito, vendar moramo biti pri tem pozorni, da pri tvorjenih besedilih ne iščemo popolnega skladanja s korpusom, saj to ni cilj NLG sistemov. Naravni jezik omogoča, da lahko v večini primerov posamezen pomen izrazimo na več različnih načinov [45].

Na področju strojnega prevajanja je bil razvit sistem BLEU (iz katerega izhajata tudi sistema NIST in ROGUE), ki tvorjeno besedilo oceni na podlagi števila vsebovanih n-gramov, ki se pojavijo tudi v enem izmed možnih prevodov. [3], [35]

Ker se pri tvorjenju besedil s skritimi markovskimi modeli besede izbirajo izključno iz obstoječih besed v korpusu, bi takšen način ocenjevanja za tvorjeno besedilo vedno podal popolno oceno. Zato smo se, kot smo omenili v uvodu, odločili, da bomo za merilo primerjave tvorjenih besedil izbrali pogostost pojavljanja določenih besednih vrst. Izbrali smo glagole, ker pričakujemo, da konstantnost njihovega pojavljanja izmed vseh besednih vrst

najbolje opisuje naravo jezika. V povprečju pričakujemo po en glavni glagol za vsak stavek.

Našo rešitev smo primerjali z dvema drugima rešitvama iz 4. poglavja. Izbrali smo hmmlearn (4.3) in umdhmm (4.4), ker se medsebojno razlikujeta po pristopu učenja modelov, zato smo pričakovali, da se bodo razlike med modeli pokazale ravno tukaj.

Da bi ugotovili morebiten vpliv spreminjanja parametrov skritih markovskih modelov na tvorjena modela, smo se odločili, da pri vseh treh rešitvah uporabimo modele z 1, 3, 5, 8 in 12 stanji. Na koncu smo izvedli še primerjavo tvorjenih besedil z izvornim korpusom. Skupno smo torej primerjali 16 besedil.

6.1 Izbira korpusa

Učno množico smo izvlekli iz korpusa slovenskega pisnega jezika ccKRES [27], ki je označen v skladu s priporočili za zapis besedil TEI P5 [48]. XML zapis vsebuje označbe za povedi, besede, ločila itn. Besede so dodatno označene z informacijami o besednih vrstah, številu, spolu ... [13]

Korpus vsebuje raznolike besedilne vrste. Da bi se izognili neobičajnim oblikam povedi, smo izločili TV sporede, kuharske recepte in športne rezultate ... To smo storili tako, da smo izbrali izključno povedi, ki se začnejo z veliko začetnico in končajo s končnim ločilom (piko, klicajem ali vprašajem). Izločili smo tudi povedi, ki vsebujejo velike začetnice, ki niso na začetku povedi. Preostale povedi smo segmentirali na mestih, zaznamovanih z vejicami, in tako dobili segmente, ki približno ustrezajo stavkom (z izjemami, kot so vrinjeni stavki in podobno). Segmente smo nato prilagodili za učenje modelov, tako da smo velike črke pretvorili v male in izločili vsa ločila. Tako pridobljeno učno množico segmentov smo shranili v tekstovno datoteko s po enim segmentom na vrstico.

6.2 Kvantitativna analiza

Zbiranje podatkov in primerjavo modelov smo avtomatizirali s programom, ki izvede naslednje korake:

1. Iz učne množice besedil naključno izbere 5.000 vrstic.
2. Na izbranih vrsticah izmeri porazdelitev števila besed.
3. Na izbranih vrsticah izmeri pogostost pojavljanja glagolov (na podlagi označb v TEI dokumentih).
4. Izbrane vrstice uporabi za učenje modelov.
5. Vsak naučen model uporabi za tvorjenje 5.000 stavkov, katerih dolžine naključno izbere glede na podatke, pridobljene v 2. koraku.
6. Za tvorjena besedila izmeri pogostost pojavljanja glagolov (kot v 3. koraku). Meritve, katere bomo v nadaljevanju uporabili za analizo, zapiše v datoteko.

Pridobljene meritve so prikazane v tabeli 6.1.

Hipoteza 1. *Število skritih stanj modela ima statistično pomemben vpliv na pogostost pojavljanja glagolov v stavkih.*

Zanimalo nas je, ali ima število skritih stanj modela zaznaven vpliv na tvorjeno besedilo. Za vsako orodje smo opravili ločen χ^2 preizkus. Uporabili smo podatke v tabeli 6.1. Rezultati preizkusa so prikazani v tabeli 6.2. Glede na visoke p vrednosti hipoteze za nobeno od orodij ne moremo potrditi. Iz tega sledi, da število skritih stanj modela nima zaznavnega vpliva na tvorjeno besedilo.

Hipoteza 2. *Pogostost pojavljanja glagolov v stavkih pri tvorjenih besedilih je primerljiva s pogostostjo pojavljanja glagolov v korpusu.*

		Število glagolov na stavek					
Orodje	**N**	0	1	2	3	4	≥ 5
hmmlearn	1	1877	1731	906	331	106	49
	3	1923	1671	873	369	117	47
	5	1921	1707	884	336	95	57
	8	1897	1737	850	340	121	55
	12	1876	1742	889	355	98	40
UMDHMM	1	2122	1663	810	282	77	46
	3	2092	1699	813	280	87	29
	5	2099	1728	783	281	74	35
	8	2088	1663	820	318	78	33
	12	2070	1703	808	283	91	45
Lastna impl.	1	1867	1757	873	341	111	51
	3	1879	1740	890	339	112	40
	5	1826	1762	871	378	109	54
	8	1854	1716	928	323	127	52
	12	1900	1693	927	342	95	43
Korpus	/	1113	2680	1019	164	18	6

N . . . število skritih stanj modela

Tabela 6.1: Frekvenca pojavljanja glagolov pri besedilih, tvorjenih z različnimi modeli, skupaj s frekvenco pojavljanja glagolov v korpusu.

Model	χ^2	**p**
UMDHMM	15,840	0,726
hmmlearn	16,399	0,692
Lastna impl.	19,625	0,482

Tabela 6.2: χ^2 in p vrednosti pri spreminjanju št. stanj modelov. za vsako orodje.

model	**N**	χ^2	**p**
hmmlearn	1	1781.45	0.0
	3	2071.20	0.0
	5	1901.00	0.0
	8	2090.52	0.0
	12	1638.61	0.0
UMDHMM	1	1888.47	0.0
	3	1696.58	0.0
	5	1664.18	0.0
	8	1745.01	0.0
	12	1858.63	0.0
Lastna impl.	1	1858.63	0.0
	3	1743.51	0.0
	5	1914.00	0.0
	8	2015.09	0.0
	12	1679.03	0.0

N . . . št. skritih stanj modela

Tabela 6.3: Primerjava χ^2 in p vrednosti za vsa tvorjena besedila.

Zanimalo nas je, ali je kateri od modelov sposoben tvorjenja besedil, ki bi se po pogostosti pojavljanja glagolov prilegala korpusu. Za vsako tvorjeno besedilo smo opravili χ^2 preizkus neodvisnosti od korpusa. Rezultati so prikazani v tabeli 6.3. Glede na p vrednosti ugotavljamo, da hipoteze za nobenega od modelov ne moremo potrditi. Iz tega sledi, da noben izmed modelov ni tvoril besedila, ki bi bilo po pogostosti pojavljanja glagolov v stavkih primerljivo s korpusom.

Hipoteza 3. *Lastna implementacija tvori besedila, ki so po pogostosti pojavljanja glagolov v stavkih primerljiva ostalim orodjem.*

Zanimalo nas je, ali so besedila, tvorjena z lastno implementacijo, pri-

Št. glagolov	**0**	**1**	**2**	**3**	**4**	**≥ 5**	$\sum$
Korpus	1113,0	2680,0	1010,0	164,0	18,0	6,0	5000
UMDHMM	2094,2	1691,2	806,8	288,8	81,4	37,6	5000
Lastna impl.	1865,2	1733,6	897,8	344,6	110,8	48,0	5000
hmmlearn	1898,8	1717,6	880,4	346,2	107,4	49,6	5000

Tabela 6.4: Porazdelitev glagolov pri izvornem besedilu in pričakovana porazdelitev (povprečje) glagolov za posamezna orodja.

	hmmlearn		UMDHMM		Lastna impl.	
	χ^2	p	χ^2	p	χ^2	p
hmmlearn	-	-	48.901	0.0	1.255	0.939
UMDHMM	45.381	0.0	-	-	57.466	0.0
Lastna impl.	1.254	0.94	60.645	0.0	-	-

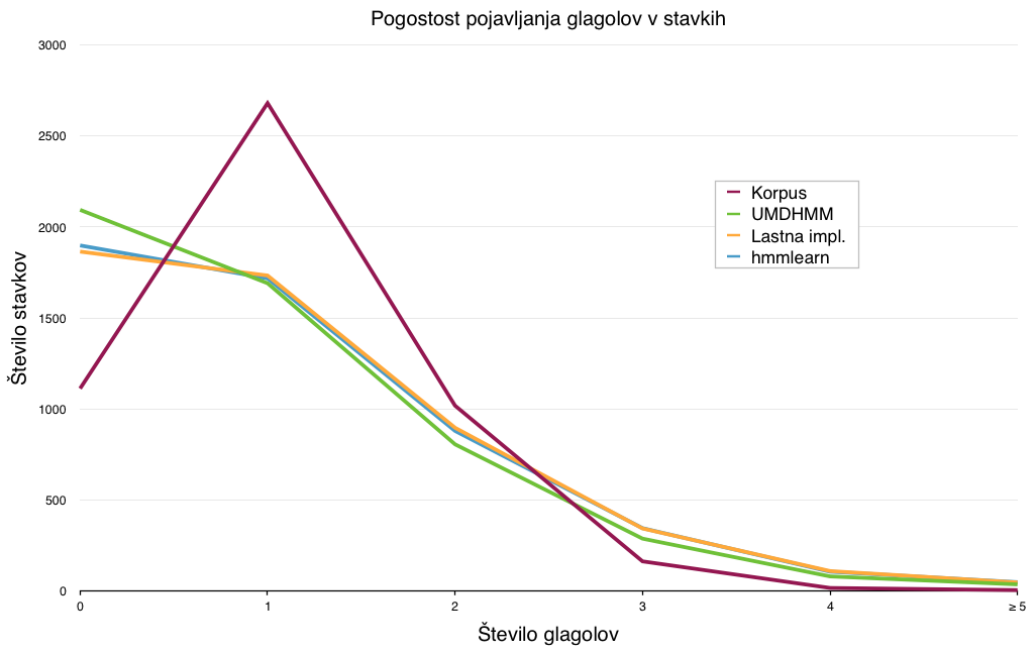
Tabela 6.5: χ^2 in p vrednosti preizkusov neodvisnosti med orodji.

merljiva z besedili, tvorjenimi z drugimi orodji. Ker smo pri 1. hipotezi ugotovili, da število skritih stanj modela nima zaznavnega vpliva, smo v tem koraku vsak model predstavili s povprečnimi vrednostmi, pridobljenimi čez vsa števila skritih stanj. Podatki so prikazani v tabeli 6.4.

Na podlagi rezultatov χ^2 preizkusov neodvisnosti v tabeli 6.5 lahko hipotezo potrdimo za našo implementacijo in orodje hmmlearn. Iz tega lahko sklepamo, da omenjeni rešitvi tvorita besedila s podobno pogostostjo pojavljanja glagolov v stavkih.

Rezultati za orodje UMDHMM kažejo veliko mero neodvisnosti od ostalih dveh orodij. Menimo, da je to posledica neskladanja v pogostosti pojavljanja stavkov brez glagolov (2. stolpec v tabeli 6.4). Pri enem in več glagolov na stavek so si vsa orodja podobna, kar potrjujejo visoke p vrednosti v tabeli 6.2.

Podatke iz tabele 6.4 smo prikazali tudi na grafu na sliki 6.1.



Slika 6.1: Pogostost pojavljanja glagolov pri različnih modelih.

6.3 Kvalitativna analiza lastne implementacije

S kvantitativno analizo smo pokazali, da naša implementacija tvori besedila, ki so po pogostosti pojavljanja glagolov v povedih (v primerjavi s korpusom) vsaj tako dobra kot ostali dve orodji. Poleg tega menimo, da je naša knjižnica temeljito dokumentirana in da pri uporabi dovoljuje veliko mero prostosti in fleksibilnosti.

- URL naslov knjižnice: `http://github.com/mfilej/himamo`
- URL naslov dokumentacije: `http://hexdocs.pm/himamo`

Poglavje 7

Sklepne ugotovitve

V diplomskem delu smo predstavili probleme, ki se pojavljajo pri uporabi skritih markovskih modelov v praksi. Njihove rešitve smo zapisali v obliki psevdokode. Predstavili smo tudi nadgradnje rešitev, ki omogočajo tvorjenje besedil s skritimi markovskimi modeli. Pregledali smo obstoječe rešitve za delo s skritimi markovskimi modeli in njihovo ustreznost za uporabo na področju tvorjenja besedil. Implementirali smo knjižnico za delo s skritimi markovskimi modeli. Postopek implementacije smo opisali in navedli tudi težave, ki se pri tem pojavljajo (npr. napaka podkoračitve). Nekaj izmed rešitev smo skupaj z lastno implementacijo preizkusili pri učenju na podlagi izbranega korpusa slovenskega pisnega jezika. Na nastalih besedilih smo opravili statistično analizo na podlagi pogostosti pojavljanja glagolov v stavkih.

Prišli smo do naslednjih ugotovitev:

1. Število skritih stanj v skritem markovskem modelu nima znatnega vpliva na tvorjeno besedilo.

2. Nobeno izmed orodij ni sposobno tvorjenja besedil, ki bi bila podobna izvornemu korpusu.

3. Lastna implementacija je sposobna tvoriti besedila s podobnimi karakteristikami kot ostala orodja.

Tvorjena besedila imajo zaradi (statistične) narave tvorjenja zaporedij besed le redko smisel. Zaradi tega menimo, da skriti markovski modeli sami po sebi niso dovolj zmogljivi za tvorjenje naravnih besedil na podlagi splošnega pisnega jezika, ki smo ga uporabili v diplomskem delu. Zaradi sposobnosti proizvajanja velikega števila variacij v besedilih pa bi lahko bili uporabni pri izbiri besed kot del širšega sistema za tvorjenje naravnega jezika.

V prihodnosti želimo skrite markovske modele uporabiti pri pripravi besedil za predmet digitalna forenzika. Pričakujemo, da bo uporaba na ožji problemski domeni privedla do uporabnih rezultatov.

Literatura

[1] J. Armstrong, *Programming Erlang: software for a concurrent world.* Pragmatic Bookshelf, 2007.

[2] K. Beck, *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[3] A. Belz in E. Reiter, "Comparing automatic and human evaluation of NLG systems", *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics*, str. 313–320, 2006.

[4] J. Bilmes, "A Gentle Tutorial of the EM algorithm and its application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models", ICSI, Teh. poročilo TR-97-021, 1997.

[5] C. Bishop, *Pattern recognition and machine learning.* Springer, 2006, ISBN: 978-0-387-31073-2.

[6] A. Bruen, *Cryptography, information theory, and error-correction : a handbook for the 21st century.* Hoboken, N.J: Wiley-Interscience, 2005, ISBN: 978-0-471-65317-2.

[7] O. Cappé, "H2M: A set of MATLAB/OCTAVE functions for the EM estimation of mixtures and hidden Markov models", *URL: Http://www. tsi. enst. fr/∼ cappe/h2m*, 2001.

[8] R. Carlsson in M. Rémond, "EUnit: A Lightweight Unit Testing Framework for Erlang", v *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, zbirka ERLANG '06, New York, NY, USA: ACM, 2006, str. 1, ISBN: 1-59593-490-1. DOI: 10.1145/1159789.1159791.

[9] A. Clark, C. Fox in S. Lappin, *The handbook of computational linguistics and natural language processing*. John Wiley & Sons, 2013.

[10] S. Comino, F. M. Manenti in M. L. Parisi, "From planning to mature: On the success of open source projects", *Research Policy*, zv. 36, št. 10, str. 1575–1586, 2007.

[11] L. Determann, "Dangerous Liaisons—Software Combinations as Derivative Works? Distribution, Installation, and Execution of Linked Programs Under Copyright Law, Commercial Licenses, and the GPL", *Berkeley Technology Law Journal*, zv. 21, št. 4, str. 1421–1498, 2006, ISSN: 10863818.

[12] A. Dufaux, L. Besacier, M. Ansorge in F. Pellandini, "Automatic sound detection and recognition for noisy environment", v *Signal Processing Conference, 2000 10th European*, IEEE, 2000, str. 1–4.

[13] T. Erjavec, C. Krstev, V. Petkevič, K. Simov, M. Tadić in D. Vitas, "The MULTEXT-east Morphosyntactic Specifications for Slavic Languages", v *Proceedings of the 2003 EACL Workshop on Morphological Processing of Slavic Languages*, zbirka MorphSlav '03, Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, str. 25–32.

[14] *ExUnit - ExUnit v1.2.5*. spletni naslov: http://elixir-lang.org/docs/stable/ex_unit/ExUnit.html (pridobljeno 25. maj 2016).

[15] G. Fele-Žorž, "An automatic disk image generator for teaching computer forensics (članek v nastajanju)",

[16] L. Gyergyék, *Teorija o informacijah*, Slovenian. Ljubljana: Fakulteta za elektrotehniko, 1988.

[17] P. Hudak, "Conception, evolution, and application of functional programming language", *ACM Computing Surveys*, zv. 21, št. 3, str. 359–411, 1989.

[18] J. Huffman, *Dialyxir*. spletni naslov: `https://github.com/jeremyjh/dialyxir` (pridobljeno 24. maj 2016).

[19] J. Hughes, "Why functional programming matters", *The computer journal*, zv. 32, št. 2, str. 98–107, 1989.

[20] S. Jurić, *Elixir in action*. Manning Publ., 2015.

[21] T. Kanungo, *UMDHMM: Hidden Markov Model Toolkit*, 1999. spletni naslov: `http://www.kanungo.com/software/software.html` (pridobljeno 18. avgust 2016).

[22] B. Knab, B. Wichern, B. Steckemetz in A. Schliep, *GHMM: General Hidden Markov Model library*. spletni naslov: `http://ghmm.sourceforge.net` (pridobljeno 30. junij 2016).

[23] R. Kondadadi, B. Howald in F. Schilder, "A Statistical NLG Framework for Aggregated Planning and Realization.", v *ACL (1)*, 2013, str. 1406–1415.

[24] S. Krek, "Od SSKJ do spletnega portala standardne slovenščine", *Jezik in slovstvo*, zv. 54, št. 3-4, str. 95–113, 2009.

[25] H. Li in S. Thompson, "Testing Erlang refactorings with quickcheck", v *Implementation and Application of Functional Languages*, Springer, 2007, str. 19–36.

[26] X. L. X. Li, M. Parizeau in R. Plamondon, "Training hidden Markov models with multiple observations-a combinatorial method", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, zv. 22, št. 4, str. 371–377, 2000, ISSN: 0162-8828. DOI: `10.1109/34.845379`.

[27] N. Logar, T. Erjavec, S. Krek, M. Grčar in P. Holozan, *Written corpus ccKres 1.0*, 2013. spletni naslov: `http://hdl.handle.net/11356/1034` (pridobljeno 18. avgust 2016).

[28] M. Luštrek, "Luščenje podatkov s skritimi markovskimi modeli", 2004.

[29] F. Mairesse, M. Gašić, F. Jurčíček, S. Keizer, B. Thomson, K. Yu in S. Young, "Phrase-based statistical language generation using graphical models and active learning", v *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, 2010, str. 1552–1561.

[30] T. P. Mann, "Numerically Stable Hidden Markov Model Implementation", *An HMM scaling tutorial*, str. 1–8, 2006.

[31] *Map - Elixir v1.2.5*. spletni naslov: `http://elixir-lang.org/docs/stable/elixir/Map.html` (pridobljeno 24. maj 2016).

[32] *No License - Choose a License*. spletni naslov: `http://choosealicense.com/no-license/` (pridobljeno 2. junij 2016).

[33] J. O'Connell in S. Højsgaard, "Hidden Semi Markov Models for Multiple Observation Sequences: The mhsmm Package for R", *Journal of Statistical Software*, zv. 39, št. 4, str. 1–22, 2011.

[34] M. Papadakis in K. Sagonas, "A PropEr integration of types and function specifications with property-based testing", v *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, ACM, 2011, str. 39–50.

[35] K. Papineni, S. Roukos, T. Ward in W. Zhu, "BLEU: a method for automatic evaluation of machine translation", *... of the 40Th Annual Meeting on ...*, št. July, str. 311–318, 2002, ISSN: 00134686. DOI: `10.3115/1073083.1073135`.

[36] parroty, *ExCheck*. spletni naslov: `https://github.com/parroty/excheck` (pridobljeno 25. maj 2016).

[37] N. Pavešić, *Informacija in kodi*, 2. izd. Ljubljana: Založba FE in FRI, 2010, ISBN: 978-961-243-145-7.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot in E. Duchesnay, "Scikit-learn: Machine Learning in Python", *Journal of Machine Learning Research*, zv. 12, str. 2825–2830, 2011.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot in É. Duchesnay, "Scikit-learn: Machine Learning in Python", *J. Mach. Learn. Res.*, zv. 12, str. 2825–2830, november 2011, ISSN: 1532-4435.

[40] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition", zv. 77, št. 2, str. 257–286, 1989, ISSN: 15582256. DOI: 10.1109/5.18626. arXiv: arXiv:1011.1669v3.

[41] A. Rahimi, *An Erratum for "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition"*. spletni naslov: http://alumni.media.mit.edu/~rahimi/rabiner/rabiner-errata/rabiner-errata.html (pridobljeno 27. maj 2016).

[42] D. Ramage, *Hidden Markov Models Fundamentals*, 2007.

[43] P. Ramesh in J. G. Wilpon, "Modeling state durations in hidden Markov models for automatic speech recognition", v *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, zv. 1, marec 1992, 381–384 vol.1. DOI: 10.1109/ICASSP.1992.225892.

[44] K. Sagonas in D. Luna, "Gradual typing of erlang programs: a wrangler experience", v *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, ACM, 2008, str. 73–82.

[45] R. Sambaraju, E. Reiter, R. Logie, A. Mckinlay, C. Mcvittie, A. Gatt in C. Sykes, "What is in a text and what does it do: Qualitative Evaluations of an NLG system – the BT-Nurse – using content analysis and discourse analysis", št. September, str. 22–31, 2011.

[46] A. Schliep, B. Georgi, W. Rungsarityotin, I. Costa in A. Schonhuth, "The general hidden markov model library: Analyzing systems with unobservable states", *Proceedings of the Heinz-billing-price*, zv. 2004, str. 121–135, 2004.

[47] S. Sonnenburg, M. L. Braun, C. S. Ong, S. Bengio, L. Bottou, G. Holmes, Y. LeCun, K.-R. Mueller, F. Pereira, C. E. Rasmussen, G. Raetsch, B. Schoelkopf, A. Smola, P. Vincent, J. Weston in R. Williamson, "The need for open source software in machine learning", *Journal of Machine Learning Research*, zv. 8, str. 2443–2466, 2007.

[48] C. M. Sperberg-McQeen, L. Burnard in S. Bauman, *TEI P5: Guidelines for electronic text encoding and interchange*, 2008. spletni naslov: `http://www.tei-c.org/Guidelines/P5/` (pridobljeno 18. avgust 2016).

[49] K. J. Stewart, A. P. Ammeter in L. M. Maruping, "Impacts of License Choice and Organizational Sponsorship on User Interest and Development Activity in Open Source Software Projects", *Information Systems Research*, zv. 17, št. 2, str. 126–144, 2006. DOI: `10.1287/isre.1060.0082`.

[50] D. Thomas, *Programming Elixir*. Pragmatic Bookshelf, 2014.

[51] S. Van Der Walt, S. C. Colbert in G. Varoquaux, "The NumPy array: a structure for efficient numerical computation", *Computing in Science & Engineering*, zv. 13, št. 2, str. 22–30, 2011.

[52] J. Van Gael, Y. Saatci, Y. W. Teh in Z. Ghahramani, "Beam sampling for the infinite hidden Markov model", v *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, str. 1088–1095.

[53] G. Varoquaux in S. Lebedev, *Hmmlearn*. spletni naslov: `https://github.com/hmmlearn/hmmlearn` (pridobljeno 30. maj 2016).

[54] L. Xu in M. I. Jordan, "On convergence properties of the EM algorithm for Gaussian mixtures", *Neural computation*, zv. 8, št. 1, str. 129–151, 1996.

[55] J. Yun, "A MATLAB toolbox to identify RNA-protein binding sites in HITS-CLIP", 2013.

[56] J. Zhao, "Hidden Markov Models with Multiple Observation Processes", Doktorska disertacija, The University of Melbourne, 2007.

[57] H. Zhou, C. Zhang, S. Liu in Y. Zhou, "Web-based toolkits for topology prediction of transmembrane helical proteins, fold recognition, structure and binding scoring, folding-kinetics analysis and comparative analysis of domain combinations", eng, zv. 33, št. Web Server issue, W193–7, julij 2005, ISSN: 0305-1048 (Print). DOI: 10.1093/nar/gki360.

[58] G. Zyskind, *HMM*. spletni naslov: https://github.com/guyz/HMM (pridobljeno 25. maj 2016).