Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Maks Horvat

Orodja za tekstovno rudarjenje v slovenščini

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: prof. dr. Marko Robnik-Šikonja

Ljubljana 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00355/2012
Datum: 05.11.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MAKS HORVAT**

Naslov: **ORODJA ZA TEKSTOVNO RUDARJENJE V SLOVENŠČINI**
TEXT MINING TOOLS FOR SLOVENE LANGUAGE

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Ogromno podatkov in informacij najdemo v obliki prostih besedil, kot so knjige, članki, navodila, novice, blogi itd. Računalniške tehnike za obdelavo naravnega jezika so način, kako iz teh besedil pridobivamo strukturirane informacije. Kot predpripravo za tovrstno napredno analizo potrebujemo dovolj velike besedilne zbirke in na njih sloneča orodja za členjenje, lematizacijo, stavčno označevanje, klasifikacijo in grupiranje teksta. Čeprav je struktura teh orodij večinoma neodvisna od konkretnega jezika, jih je vendarle potrebno prilagoditi posameznim jezikovnim in statističnim značilnostim posameznega jezika.
Preglejte stanje na področju jezikovnih pripomočkov za slovenski jezik in pripravite nekaj orodij, ki bodo delovala v programskem jeziku python. Pokažite, kako lahko s temi orodji rešite nekaj tipičnih nalog, ki se pojavljajo pri tekstovnem rudarjenju.

Mentor:
prof. dr. Marko Robnik Šikonja

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:
prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Maks Horvat, z vpisno številko **63060190**, sem avtor diplomskega dela z naslovom:

Orodja za tekstovno rudarjenje v slovenščini

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Marka Robnik-Šikonje,
- 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 v zbirki "Dela FRI".

V Ljubljani, dne 13. marec 2013 Podpis avtorja:

Zahvaljujem se prof. dr. Marku Robnik Šikonji za nasvete in usmeritvi pri izdelavi diplomske naloge.

Zahvaljujem se svojim staršem za podporo in možnost mirnega študija.

Zahvaljujem se prijateljem za dogodivščine tekom študija.

Zahvaljujem se dekletu za podporo pri izdelavi diplomske naloge.

Kazalo

Slike

Tabele

Povzetek

Cilj diplomske naloge je bil predstaviti rabo različnih orodij za obdelavo slovenskega jezika in jih prilagoditi za delovanje v knjižnici NLTK. Reševali smo problem avtomatskega določanja besednih vrst in uporabili algoritme iz knjižnice NLTK. Iz korpusa Gigafida smo zgradili označevalnike n-gram, Brill, Naive Bayes, maksimalna entropija in skriti markovski model. Za navedene označevalnike smo izmerili natančnost oblikoslovnega označevanja in časovno kompleksnost ter primerjali dobljene rezultate med seboj. V NLTK smo vključili orodje Obeliks za lematizacijo in oblikoslovno označevanje besedil. Za razčlenjevanje besedil in razpoznavanje imenskih entitet smo uporabili orodji dependecy parser in slner. Razvili smo orodje za poizvedovanje po informacijah in ga testirali na skupini dokumentov. Pri tem smo uporabljali: obrnjeni indeks, logične izraze, vektorski model in kosinusno podobnost.

Abstract

We introduce the use of various tools for Slovenian language processing and adapt them for NLTK library. To automatically determine the part of speech tags we use algorithms from the NLTK library. From Gigafida corpus we build several taggers: n-gram, Brill, naive Bayes, maximum entropy and hidden Markov model. We measure the accuracy of part of speech tags and time complexity of the taggers. We also incorporated Obeliks program for lemmatization and part of speech tags assignment. For text parsing and identification of named entities we use dependencyParser and SLNER tools. We develop and test a module for information retrieval. We use inverted index, search with boolean operators, vector representation of documents and cosine similarity.

Poglavje 1

Uvod

Procesiranje naravnega jezika se ukvarja z eno od najstarejših človeških iznajdb, človeškim jezikom. Pisni jezik najdemo v različnih oblikah: e-pošta, spletne strani, opisi izdelkov, zgodbe v časopisih, socialni mediji, znanstveni članki, knjige,... V zadnjem desetletju so uspešne aplikacije za obdelavo naravnega jezika postale del našega vsakdana, od črkovalnikov in slovničnih programov, strojnega prevajanja besedil na spletu, odkrivanje neželene pošte, odkrivanje mnenja ljudi o izdelkih ali storitvah,...

Procesiranje naravnega jezika (ang. "Natural Language Procssing" oz. NLP) je raziskovalno področje, ki se ukvarja z obdelavo nestrukturiranih besedil zapisanih v naravnem jeziku. Namen NLP je obdelati besedilo tako, da ga bo računalnik razumel kot bi ga človek, kar je izjemno težko. Takim problemom pravimo, da so najtežji problemi s področja umetne inteligence (AI - artificial inteligence). Alen Turing je razmišljal o metodi, ki bi v praksi razlikovala umetno inteligenco od naravne. Pripravil je sklop vprašanj in odgovorov, ki si jih prek tipkovnice izmenjamo z dvema osebkoma, da bi ugotovili, ali se pogovarjamo s strojem ali s človekom. Turingov test, kot je bil pozneje poimenovan, je uspešno rešen, če razsodnik ne razloči, ali se pogovarja s človekom ali s strojem. CAPTCHA (Completely Automated Public Turing Test to Tell Computers and Humans Apart) je poskus avtomatiziranega Turingovega testa, ki ne potrebuje človeškega razsodnika. Velja

omeniti, da do sedaj računalniku ni uspelo v celoti rešiti Turingov test.

V okviru diplomske naloge smo razvili nekaj orodij za slovenski jezik v knjižnici Natural Language Toolkit (krajše NLTK[1]), zasnovani v programskem jeziku Python. Pritegnila nas je enostavna uporaba, podprta s prosto dostopno, precej obsežno dokumentacijo. Od prve različice leta 2001 pa do danes se je v knjižnici zbralo več kot 100.000 vrstic kode. Številni vgrajeni moduli so napisani v hitrejših in optimiziranih jezikih, kar omogoča zmogljivost in hitrost izvajanja številnih algoritmov za procesiranje naravnega jezika. Python s pregledno in enostavno sintakso omogoča hitro pisanje kratkih in zmogljivih programov. Prednost obojega je v učenju in spoznavanju naravnega jezika skozi zmogljiva orodja, ki s pomočjo Pythona ohranijo enostavnost. NLTK se uspešno uporablja kot učni pripomoček, samostojno študijsko orodje ali platforma za raziskave prototipov in gradnje sistemov.

Knjižnica vsebuje več kot 50 korpusov z obsežno literaturo in dokumentacijo. Avtorji projekta so izvedenci na področju procesiranja naravnega jezika: Ewan Klein, Steven Bird in Edward Loper. Za hitrejše učenje so napisali knjigo Natural Language Processing with Python[2], podprto s številnimi praktičnimi primeri uporabe. Branje je zelo priporočljivo tako za začetnika kot za zahtevnejšega uporabnika. V času pisanja tega dela je bila na voljo različica NLTK 2.04, za katero je prilagojena izvorna koda v tej diplomski nalogi. Za delovanje knjižice je potrebno namestiti Python verzije 2.6-2.7. Trenutno NLTK za Python 3 še ni na voljo, saj prehod in prilagoditev še potekata.

1.1 Pregled vsebine

Diplomsko nalogo smo razdelili na osem poglavij. Drugo poglavje opisuje definicijo in pomen besedilnih korpusov, hkrati so podani vsi besedilni viri uporabljeni v našem diplomskem delu. V tretjem poglavju smo predstavili algoritme za označevanje besednih vrst in demonstrirali uporabo s kratkimi izseki programske kode. V četrtem poglavju opišemo razčlenjevanje besedil.

V petem poglavju prikažemo delovanje razpoznavanja imenskih entitet. Šesto poglavje smo namenili poizvedbam po informacijah. Opisali smo štiri najbolj pogoste tehnike in pokazali njihovo uporabo v praksi. V zadnjem poglavju prikažemo sliko uporabe posameznih modulov.

Poglavje 2

Besedilni korpusi

V našem delu uporabljamo referenčne korpuse, ki jih odlikuje velik obseg in natančna struktura, ki se odraža v ravnoteženih razmerjih med zajetimi besedili. Referenčni korpus je namenjen preučevanju jezika vsakdanje rabe in se uporablja za pridobivanje podatkov, ki so osnova za razvijanje orodij tekstovnega rudarjenja.

Besedilni korpusi so obsežne strukturirane zbirke najrazličnejših zvrsti besedil (leposlovje, stvarna literatura, strokovna literatura,...), zajetih v določenem časovnem obdobju, nastalih po vnaprej določenih merilih, z določenimi cilji in namenjeni raziskovanju naravnega jezika na več ravneh. V današnjem času so običajno shranjeni v digitalni obliki, lahko kot prosto besedilo, vendar se izkaže format XML za ustreznejšega, saj omogoča vključevanje dodatnih meta podatkov, kot so leme[1] (glej razdelek 2.1), besedne vrste (glej poglavje 3),...

[1]Lema je kanonična, osnovna oblika besede (npr. lema za besedo "hitrega" je "hiter", za "kolesarji" "kolesar" itd.)

2.1 Razlaga msd kod

Msd[23] koda je sestavljena iz več znakov, kjer vsak pove neko kategorijo besede. Najpomembnejši je prvi znak, ki pove besedno vrsto, in sicer: S pomeni samostalnik, G glagol, P pridevnik, R prislov, D predlog, V veznik, L členek, M medmet, K številnik, O okrajšava in Z zaimek. Naslednji znaki povedo dodatne lastnosti dane besdne vrste; spol, sklon, število za samostalnik; čas, osebo za glagol itd. Primer uporabe kode msd je na sliki 2.1 in 2.2. V tabeli 2.1 je primer kode msd za besedo "lep".

pridevnik	
vrsta	splošni
spol	moški
število	ednina
sklon	imenovalnik
živost	0
vid	0
oblika	0
oseba	0
nikalnost	0
stopnja	nedoločeno
določnost	ne
število_svojine	0
spol_svojine	0
naslonskost	0
zapis	0

Tabela 2.1: Primer kode msd za besedo *lep*.

2.2 Primeri slovenskih korpusov

V sklopu projekta "Sporazumevanje v slovenskem jeziku" [3] je potekala izdelava oz. posodobitev in nadgradnja več besedilnih korpusov za slovenščino. Nastalo jih je šest, med njimi korpusi pisne slovenščine Gigafida, KRES, ccGigafida, ccKres, GOS (korpus govorjene slovenščine) in Šolar. Dosegljivi so na spletnem naslovu projekta. V nadaljevanju bomo opisali tiste, ki smo jih uporabili v našem delu.

Gigafida[22] vsebuje besedila, ki so izšla med letom 1990 in 2011. Gre za tiskana besedila in za besedila, pridobljena s spletnih strani. Tiskana besedila so izšla bodisi kot knjige z leposlovno ali stvarno vsebino bodisi v periodični obliki kot revije ali časopisi. Besedila s spleta so pridobljena iz novičarskih portalov ter predstavitvenih strani večjih slovenskih podjetij in pomembnejših državnih, pedagoških, raziskovalnih, kulturnih ustanov. Vsebuje skoraj 1,2 milijarde besed.

ccGigafida[20] je uravnotežen izbor besedil iz korpusa Gigafida. Vsebuje približno 9 % korpusa Gigafida oz. 100 milijonov besed. Vsebuje zbirke slovenskih besedil najrazličnejših zvrsti, od dnevnih časopisov, revij do knjižnih publikacij vseh vrst (leposlovje, učbeniki, stvarna literatura), spletnih besedil, prepisov parlamentarnih govorov in podobno. Uporabili smo ga za učenje modelov pri označevanju besedil (glej poglavje 3).

Šolar[21] je korpus šolskih pisnih izdelkov, ki so jih učenci slovenskih osnovnih in srednjih šol samostojno tvorili pri pouku. Korpus vsebuje skoraj milijon besed z jezikovnimi popravki, narejenih s strani učiteljev.

Šolar vsebuje 2.703 pisnih besedil srednješolcev in učencev zadnjega triletja osnovnih šol, nekaj pa je tudi besedil učencev 6. razreda osnovne šole.

- Največji delež besedil predstavljajo eseji oziroma spisi (64,2 %),
- v korpusu najdemo še pisne izdelke nastale pri učni uri, kot so obnove, opisi, prošnje ipd. (18 %),
- teste, ki so razdeljeni v dve skupini: odgovori na vprašanja (16,1 %)

predstavljajo klasične teste z vprašanji in (esejskimi) odgovori, ki so nastali pri različnih predmetih,
- v skupino daljše besedilo (1,7 %) sodijo različna praktična besedila (prošnje, zahvale ...), ki so jih učenci napisali v okviru daljšega testa iz slovenščine.

```
<p xml:id="F0000002.000050">
   <s>
      <c>»</c>
      <w msd="L" lemma="skoraj">Skoraj</w>
      <S/>
      <w msd="Zc-mmi" lemma="ves">vsi</w>
      <S/>
      <w msd="Sommi" lemma="kolesar">kolesarji</w>
      <S/>
      <w msd="Ggnstm" lemma="segati">segajo</w>
      <S/>
      <w msd="Dm" lemma="po">po</w>
      <S/>
      <w msd="Sosmm" lemma="poživilo">poživilih</w>
      <c>!</c>
      <c>«</c>
   </s>
</p>
```

Slika 2.1: Primer stavka iz ccGigafida.

```
<w3 lemma="vrednota" msd="Sozmr">vrednot</w3>
<S/>
<w3 lemma="in" msd="Vp">in</w3>
<S/>
<w3 lemma="lasten" msd="Ppnser">lastnega</w3>
<S/>
<u1 tip="Z" podtip="CRR">
<w1 lemma="mišlenje" msd="Soser">mišlenja</w1>
<p1>
<w2 lemma="mišljenje" msd="Soser">mišljenja</w2>
</p1>
</u1>
<c3>,</c3>
<S/>
<w3 lemma="ampak" msd="Vp">ampak</w3>
```

Slika 2.2: Korpus Solar.

Poglavje 3

Oblikoslovno označevanje besedila

V tem poglavju bomo predstavili pomembnost oblikoslovnega označevanja za obdelavo naravnega jezika.

Oblikoslovno označevanje[1] je postopek, pri katerem besede zaporedno označimo z skladenjskimi oznakami (npr. samostalnik, pridevnik, veznik,...). Slovenski jezik spada med jezikoslovno bogatejše jezike, zato je označevanje besedil težek problem, saj mora algoritem pravilno izbrati med okoli dva tisoč oznakami. Računalniško oblikoslovno označevanje je bilo najprej razvito za angleški jezik, pri katerem je nabor oblikoslovnih oznak razmeroma majhen (okoli šestdeset). Problem so predvsem dvoumne besede, kar nazorno prikaže tabela 3.1.

[1] alternativni angleški izrazi: part-of-speech tagging, POS tagging, word-category disambiguation, grammatical tagging

gori	na	gori	gori

lema	gor	gora	goreti
msd oznaka	Rsn	Sozed	Ggnste

Tabela 3.1: Primer dvoumnosti besede *gori*

Rsn=prislov; vrsta=splošni; stopnja=nedoločeno;
Sozed=samostalnik; vrsta=občno ime; spol=ženski; število=ednina; sklon=dajalnik;
Ggnste=glagol; vrsta=glavni; število=ednina; vid=nedovršni; oblika=sedanjik; oseba=tretja;

3.1 Tehnike označevanja

Oblikoslovno označevanje izvajamo z modeli strojnega učenja, ki jih razdelimo na:

- **Nadzorovano učenje** (supervised learnig): učni algoritmi odkrivajo pravila v označenih primerih. Večinoma gre za klasikacijo podatkov, kjer se algoritmi naučijo, kateremu razredu pripada nek primer.
- **Nenadzorovano učenje** (unsupervised learning): učni algoritmi odkrivajo zakonitosti v neoznačenih podatkih. Rezultat so gruče primerov z določenimi skupnimi lastnosti.
- **Spodbujevano učenje** (reinforcement learning): algoritem se uči z nagrajevanjem in kaznovanjem.

Uporabljeni algoritmi spadajo v kategorijo nadzorovanega učenja. Glede načina delovanja jih lahko še dodatno razdelimo:

- *Transformativni modeli*: začnejo z nekaj preprostimi rešitvami, nad katerimi uporabijo transformacije. Na vsakem korake izberemo transformacijo z največ doprinosa. Algoritem se ustavi, ko izbrana transformacija ne spre-

meni podatkov na dovolj mestih ali ko ni več možnih transformacij. Eden algoritmov, ki spada v to skupino je Brillov algoritem (poglavje 3.1.2).

- *Generativni modeli*: vrnejo skupne verjetnostno porazdelitev (joint probability distribution) obeh opazovanih spremenljivk X in Y kot P(x,y). Privzamemo neodvisnost spremenljivk, da lahko uporabimo Bayesov teorem za določitev posteriorne verjetnosti in izberemo najbolj verjetno (maximum liklehood) vrednost za y. Za izbiro uteži uporabimo relativne frekvence pojavitev besed v korpusu. Te modele natančnejo predstavimo v podpoglavjih N-GRAM (3.1.1), Naive Bayes (3.1.3.1) in skriti Markovski modeli (3.1.4).

- *Diskriminativni modeli*: z njimi modeliramo pogojno verjetnostno porazdelitev p(x|y), ki jo lahko uporabimo za napovedovanje. Za razliko od generativnih modelov diskriminativni modeli ne ustvarjajo vzorcev iz skupnih porazdelitev x in y. Mnogokrat so boljši vendar počasnejši pri učenju in težji za implementacijo. Natančneje jih predstavimo v poglavju o maksimalni entropiji (3.1.3.2).

Preden začnemo opisovati tehnike označevanja, si poglejmo na sliki 3.1 kratek odsek kode za nalaganje korpusa, razdelitev na testno in učno množico ter klic označevalnika.

```
1  # čitalec za označena besedila v pos formatu. Primer "pos" formata:
2  # malo/rsn pa/vp si/zp---d--k tudi/l lahko/rsn
3  # beseda1/oznaka1 beseda2/oznaka2 beseda3/oznaka3 ...
4  # Vsaka poved je v svoji vrstici.
5  from nltk.corpus.reader import TaggedCorpusReader
6  import transformXML as transformXML
7
8  # priprava učne množice
9  # vhodna xml datoteka korpusa ccGigafida
10 input = '..\\Diploma\\Corpus\\F0000002.xml'
11 # izhodna datoteka v pos format
12 out = '..\\Diploma\\Corpus\\TrainPOS.pos'
13 # pretvorba xml datoteke korpusa ccGigafida v pos formatu
14 transformXML.transform_to_pos(input, out)
15
16 # priprava testne množice
17 # vhodna xml datoteka korpusa ccGigafida
18 input = '..\\Diploma\\Corpus\\F0000003.xml'
19 # izhodna pretvorjenja vhodna datoteka v pos format
20 out = '..\\Diploma\\Corpus\\TestPOS.pos'
21 # pretvorba xml datoteke korpusa ccGigafida v pos format
22 transformXML.transform_to_pos(input, out)
23
24 inputFiles='\\Diploma\\Corpus'
25 readerTrainer = TaggedCorpusReader(inputFiles, 'TrainPOS.pos')
26 readerTest = TaggedCorpusReader(inputFiles, 'TestPOS.pos')
27 # shrani označene povedi učne množice
28 train_sents = readerTrainer.tagged_sents()
29 # shrani označene povedi testne množice
30 test_sents = readerTest.tagged_sents()
```

Slika 3.1: Priprava na učenje modelov.

V naših primerih označevanja bomo predvidevali predhodno uporabo vrstic 24-30 s slike 3.1.

3.1.1 N-gramski označevalnik

Pri n-gramskem označevalnem modelu je določanje oznak besedam odvisno od n-1 predhodno videnih besed. Ideja modela je, da namesto računanja skupnih verjetnosti vseh besed v povedi, izberemo nekaj (n-1) zadnjih besed. Pri tem se sklicujemo na markovsko lastnost (poglavje 3.1.4), ki pravi, da je verjetnost besede odvisna samo od prejšnje besede.

Unigram ali 1-gram upošteva le trenutno besedo. Za vsako besedo določi oznako, ki se največkrat pojavi s to besedo. Na primer „domnevni“ je po msd oznakah lahko v različnih kontekstih označen kot: Ppnmeid, Ppnmetd, Ppnmmi, Ppnzed, Ppnzem, Ppnzdi, Ppnsdi, Ppnsdt. Unigram bi preštel frekvenco pojavitev besede „domnevni“ z naštetimi oznakami in izbral najpogostejšo oznako.

Bigram poleg trenutne besede upošteva še prejšnjo besedo. Tako v slovarju hrani oznake obeh besed in izbere najpogostejšo oznako za ta par. N-grami višji od ena se v samostojni uporabi obnesejo slabše kot unigrami. To je posledica dejstva, da se kombinacije besed v določenem vrstnem redu veliko redkeje pojavijo kot besede posamezno. N-gram bo dodelil privzeto oznako „None“ vsem kombinacijam besed, na katere ni naletel v času učenja modela ali v času označevanju novega besedila. To težavo rešimo tako, da uporabimo metodo „back-off“. To pomeni, da kombiniramo več redov n-grama, kar izboljša točnost. Takšno rabo lahko smatramo za zaporedni (sequential) označevalnik. Postopek je sledeč. Besede označimo z višjim redom n-grama. Primere, ki niso bili označeni, prepustimo nižjemu redu n-grama. To ponavljamo, dokler ne pridemo do privzetega (default) označevalnika, pri katerem nam preostanejo tri možnosti: uporabimo najpogostejšo oznako v času učenja n-grama (tako imamo največ možnosti, da zadanemo neznano besedo), ročno napišemo lingvistična pravila ali označimo z oznako <UNK>(unknown) in zaključimo.

N-gram izračuna verjetnosti oznake trenutno neoznačene besede z

$$P(t_i \mid t_{i-1}, t_{i-2}, ..., t_{i-n}), \tag{3.1}$$

kjer i predstavlja i-to oznako.

Primer uporabe n-gramskega označevalnika je na sliki 3.2.

```
1  import NLTK
2
3  # 3-gram zaporedni označevalnik z vračanjem na nižji red
4  # n-grama do privzetega označevalnika
5  # 'VP' oznaka je po msd kodi veznik, vrsta=priredni
6  t0 = nltk.DefaultTagger('VP')
7  t1 = nltk.UnigramTagger(train_sents,backoff=t0)
8  t2 = nltk.BigramTagger(train_sents,backoff=t1)
9  t3 = nltk.TrigramTagger(train_sents,backoff=t2)
10
11 # ocena točnosti označevalnika
12 print t 3.evaluate(test sents)
```

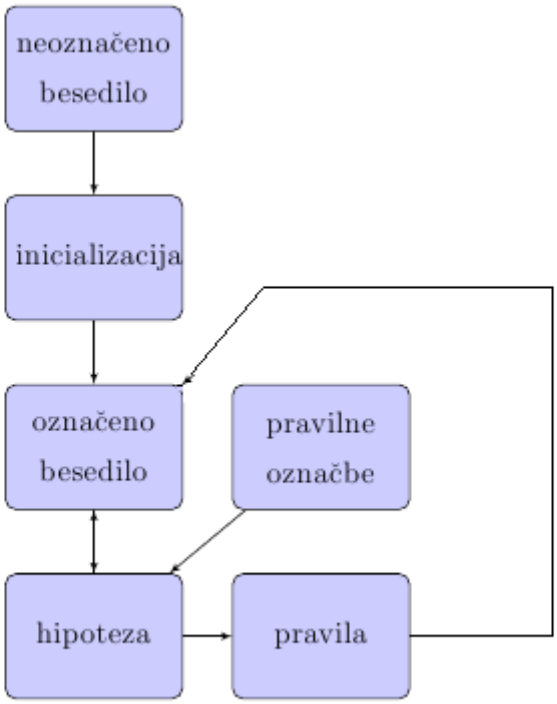
Slika 3.2: Označevanje z n-grami.

Učenje n-gramov je hitro, mnogo hitrejše kot v nadaljevanju opisani modeli. Model je preprost in razumljiv. Razširimo ga lahko na 4-grame, 5-grame,... Ker pa so oznake besed, v naravnem jeziku odvisne od vseh besed v povedi, jih iz računskega in prostorskega razloga ne uporabljamo.

3.1.2 Brillov model

Leta 1993 je Eric Brill v doktorski disertaciji[6] opisal tudi model za učenje popravkov, ki nadomestijo ročno vnašanje lingvističnih pravil. Uporablja le majhen del prostora, ki ga zahtevajo stohastični n-grami (poglavje 3.1.1).

Osnovna ideja je preprosta. Uporabi sklop pravil za označbo besedila. V primeru napak jih popravi. V tem času se nauči novih pravil, ki se uporabijo za ponovno označitev popravljenega besedila. Ta postopek se nadaljuje, dokler nova pravila povečajo točnost oznak v besedilu. Na sliki 3.3 je prikazan diagram delovanja modela.



Slika 3.3: Shema Brillovega označevalnika.

Algoritem deluje dvofazno. V prvi fazi (inicializacija) model označi besede v neoznačenem besedilu z njihovimi najverjetnejšimi oznakami. To običajno naredi z unigram označevalnim modelom. V sklopu diplome smo uporabili sekvenčni označevalnik z n-grami. V drugi fazi se na označenem besedilu uporabi množica transformacijskih pravil, kar je računsko najzahtevnejši korak. Primer uporabe Brillovega označevalnika je na sliki 3.4.

```
1  import nltk
2
3  # n-gram koda iz prejšnjega poglavja je potrebna
4  # za označitev v fazi inicializacije
5  t0 = nltk.DefaultTagger('VP')
6  t1 = nltk.UnigramTagger(train_sents,backoff=t0)
7  t2 = nltk.BigramTagger(train_sents,backoff=t1)
8  t3 = nltk.TrigramTagger(train_sents,backoff=t2)
9
10 # primer transformacijskih pravil
11 templates = [
12 nltk.brill.SymmetricProximateTokensTemplate(
13     nltk.brill.ProximateTagsRule, (1,1)),
14 nltk.brill.SymmetricProximateTokensTemplate(
15     nltk.brill.ProximateTagsRule, (2,2)),
16 nltk.brill.SymmetricProximateTokensTemplate(
17     nltk.brill.ProximateTagsRule, (1,2)),]
18
19 trainer = nltk.brill.FastBrillTaggerTrainer(t3, templates)
20 braubt_tagger = trainer.train(train_sents, max_rules=100, min_score=1)
21
22 # ocena točnosti označevalnika
23 print '%.2f%%' % (100 * t3.evaluate(test_sents))
```

Slika 3.4: Označevanje z modelom Brill.

3.1.3 Klasifikatorji

Naloga klasifikatorja je za objekt, opisan z množico značilk, določiti kateremu izmed možnih razredov pripada. Značilke so neodvisne zvezne ali diskretne spremenljivke, s katerimi opišemo primer, razred pa je odvisna diskretna spremenljivka, ki ji napovedujemo vrednost (izberemo razred). Zato, da lahko klasifikator določi razred, mora preslikati prostor atributov v razred. Preslikava je lahko podana vnaprej ali pa je naučena iz podatkov. Naloga učnega algoritma je torej iz množice vzorcev z znanimi razredi zgraditi pravilo, ki ga lahko uporabimo pri klasifikaciji. Poleg spodaj opisanih klasifikatorjev obstajajo še mnogi drugi.

Naivni Bayesov klasifikator

Poimenovan je po Thomasu Bayesu in deluje na podlagi enačbe izpeljane iz Bayesovega teorema. Naj bo y spremenljivka, ki pripada množici k razredov $\{c_1, \ldots, c_k\}$. Verjetnost razreda y pri danem x je enaka[7]:

$$P(y \mid x) = P(y) \times P(x \mid y), \tag{3.2}$$

kjer je P(y) apriorna verjetnost razreda y in $P(x \mid y)$ verjetnost oblikoslovne oznake x pri dani besedi y. Ob predpostavki pogojne neodvisnosti atributov (zaradi te predpostavke je algoritem označen kot naivni) velja[7]:

$$P(x \mid y) \cong \prod_{i=1}^{n} P(x_i \mid y), \tag{3.3}$$

Pri izbiranju oznak za besede naivni Bayes izračuna apriorno vrednost vsake oznake tako, da preveri pogostost pojavitve pri določeni besedi z izbranimi značilkami. Verjetnost oznake za besedo dobimo tako, da prispevke vseh značilk zmnožimo. Za naš problem, ki je več razredna klasifikacija, iščemo oznako oziroma ciljni razred z največjo verjetnostjo (maximum likelihood). Ta je dodeljena neoznačeni besedi.

$$t_i = \arg\max_{y} \left(P(x \mid y) = \prod_{i=1}^{n} P(x_i \mid y) \right), \tag{3.4}$$

Naivni Bayes je relativno precej odporen na šum (napačne označbe, manjkajoče besede,...). Primer uporabe naivnega Bayesa je na sliki 3.5.

```
1  import nltk
2  import NaiveBayesTagger
3
4  # učenje Naivnega Bayesa
5  naive_Bayes=NaiveBayesTagger.ClassifierBasedPOSTagger(
6  train=train_sents,verbose=True)
7
8  # ocena točnosti
9  print '%.2f%%' % (100 * naive_Bayes.evaluate(test_sents))
```

Slika 3.5: Označevanje z modelom Naive Bayes.

Označevanje z maksimalno entropijo

Maksimalni entropijski model (maximum entropy) je v NLP enak večrazredni logistični regresiji v statistiki in strojnem učenju. Pri analizi naravnega jezika sto tisoč ali milijon značilk ni nič posebnega.

Entropija[8] ali Shannonova entropija (po matematiku Claude Elwood Shannon) je količina, ki meri negotovost izzida poskusa povezanega s slučajno spremeljivko. Določa pričakovano količino informacije, ki jo pridobimo, ko izvedemo poskus in dobimo izid. Entropija je torej merilo za količino informacije, ki jo dobimo s poznavanjem vrednosti slučajne spremenljivke. Primer računanja entropije v NLTK je na sliki 3.6.

```
import math
def entropy(labels):
    freqdist = nltk.FreqDist(labels)
    probs = [freqdist.freq(l) for l in nltk.FreqDist(labels)]
    return -sum([p * math.log(p,2) for p in probs])
```

```
>>> print entropy(['male', 'male', 'male', 'male'])
0.0
>>> print entropy(['male', 'female', 'male', 'male'])
0.811278124459
>>> print entropy(['female', 'male', 'female', 'male'])
1.0
>>> print entropy(['female', 'female', 'male', 'female'])
0.811278124459
>>> print entropy(['female', 'female', 'female', 'female'])
0.0
```

Slika 3.6: Primer računanja entropije.

Ideja označevanja z maksimalno entropijo je poiskati tako verjetnostno distribucijo, ki bo izpolnjevala vse pogoje izbranih omejitev, dobljenih z izbranimi značilkami. Želimo izbrati razporeditev najbljižjo uniformni z največjo entropijo. V tabeli 3.2 je v vsaki vrstici možna oblikoslovna oznaka z msd kodo (poglavje 2.1) besede "domnevni". Oznake A, B, C,... predstavljajo značilke za vsako oblikoslovno oznako. To je lahko število pojavitev besede "domnevni", prejšnja oznaka ali beseda, beseda z veliko ali malo začetnico, predpona in končnica besede,...

	A	B	C	D	E	F	G
Ppnmeid	10%	9%	10%	8%	10%	10%	10%
Ppnmetd	5%	15%	0%	30%	0%	8%	12%
Ppnmmi	0%	100%	0%	0%	1%	0%	0%

Tabela 3.2: Primer verjetnostnih distribucij možnih oblikoslovnih oznak besede "domnevni".

Navedene distribucije so vse pravilne, vendar po principu maksimalne entropije iščemo tisto, ki je najbližja uniformni porazdelitvi. Sledimo Occamovi

britvi (Occam's razor), ki pravi, če imamo več pravilnih hipotez izberemo preprostejšo, oziroma tisto, ki je razumljivejša.

Predpostavimo, da imamo n podanih značilk označenih kot frekvence f_i, ki opredeljujejo statistično porazdelitev za modeliranje procesa. Želimo, da p leži v podmnožici C definirani z:

$$C \equiv \left\{ p \in P \mid p(f_i) = \tilde{p}(f_i) \quad for\ i \in \{1, 2, ..., n\} \right\} \tag{3.5}$$

- p (teoretična verjetnost): delež danega dogodka med vsemi možnimi dogodki,
- $\tilde{p}$ (empirična verjetnost): verjetnost pojavljanja danega dogodka med vsemi dogodki/izidi, ki smo jih opazili,
- P predstavlja množico vseh verjetnostnih porazdelitev omejenih z značilkami. Omejitev predstavlja relacijo med pričakovano vrednostjo funkcije značilk v modelu in njihovimi pričakovanimi vrednostmi v učnih podatkih (v našem primeru v učnem korpusu). V bistvu je to preslikava teoretične verjetnosti v empirično verjetnost značilk.

Ideja maksimalne entropije narekuje, da med modeli $p \in C$, izberemo tistega z najbolj uniformno porazdelitvijo. Problem uniformnosti pogojne distribucije $p(y \mid x)$ rešujemo z enačbo pogojne entropije:

$$H(p) \equiv -\sum_{x,y} \tilde{p}(x) p(y|x) \log p(x|y) \tag{3.6}$$

Po principu maksimalne entropije za izbiro modela iz množice C dovoljenih verjetnostnih porazdelitev, izberemo model $p* \in C$ z maksimalno entropijo H(p):

$$p* = \arg\max_{p \in C} H(p) \tag{3.7}$$

Enačbo rešujemo z različnimi optimizacijskimi algoritmi. V NLTK lahko uporabimo naslednje algoritme: GIS, IIS, CG, BFGS, Powell, LBFGSB,

Nelder-Mead, MEGAM[25][2] in TADM[26] [3]. Čas učenja in točnosti naučenih modelov se občutno razlikujejo glede na izbiro algoritma. GIS (Generalized Iterative Scaling) in IIS (Improved Iterative Scaling) ne potrebujeta dodatnih knjižnic in sta najpočasnejša med naštetimi. Inštalacija knjižice Scipy[27][4] je potrebna za agoritme CG (Conjugate Gradient), BFGS (Broyden-Fletcher-Goldfarb-Shanno), Powell, LBFGSB (z delovnim pomnilnikom omejena varianta BFGS) in Nelder-Mead. V novejših različicah je modul maxentropy odstranjen, ovijalni modul nltk.classify.maxent pa še ni ustrezno prilagojen, pričakovane so spremembe v naslednjih različicah NLTK. Razvijalci priporočajo uporabo logistične regresije v scikit-learn[28][5]. Za večrazredne klasifikacijske probleme, kot je označevanje besed, najbolj deluje L-BFGS, oziroma paketi z njegovimi implementacijami. Primer uporabe je na sliki 3.7.

[2]MEGAM uporablja CG in L-BFGS tehnike za optimiziranje logistične regresije.

[3]TADM (Toolkit for advanced Discriminative Modeling) je v c++ implementirana zunanja knjižnica za ocenjevanje parametrov diskriminantnih modelov.

[4]Scipy je odprto kodna knjižnica raznih matematičnih orodij namenjena uporabi v programskem jeziku Python. Za delovnje je potrebno namestitev Numpy.

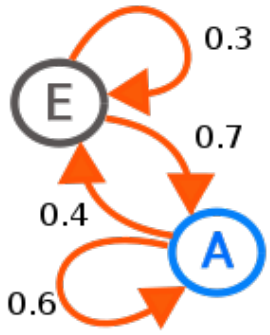
[5]Scikit-learn je odprto kodna knjižnica algoritmov strojnega učenja. Za delovanje potrebuje namestitev Scipy in Numpy.

```
27  import nltk
28
29
30  # Učenje maksimalne entropije z algoritmom GIS. Uporabili smo lambda
31  # funkcijo za notranji klic modula maksimalne entropije. Število
32  # iteracij algoritma sta 2, kar je zelo malo in se tudi pozna pri
33  # točnosti, vendar se pri obdelavi velikih korpusov z vsako iteracijo
34  # krepko poveča čas učenja modela. Načeloma iščemo dobro razmerje
35  # med časom učenja in točnostjo modela.
36  maxentTagger = nltk.ClassifierBasedPOSTagger(train=train_sents,
37      classifier_builder=lambda train_feats:
38      nltk.MaxentClassifier.train(train_feats, algorithm=model,
39      max_iter=iter, min_lldelta=0.1))
40
41  # Ocena točnosti modela
42  print '%.2f%%' % (100 * maxentTagger.evaluate(test_sents))
```

Slika 3.7: Označevanje z maksimalno entropijo.

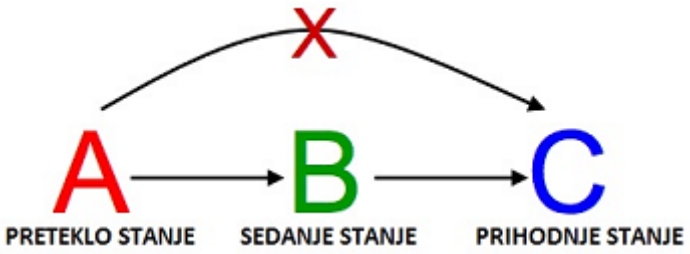
3.1.4 Skriti markovski model

Najenostavnejši markovski model je markovska veriga. Modelira stanje sistema z slučajnimi spremenljivkami, ki se spreminjajo skozi čas. Markovski proces z diskretnim prostorom stanj in diskretno časovno množico imenujemo markovska veriga (slika 3.8).

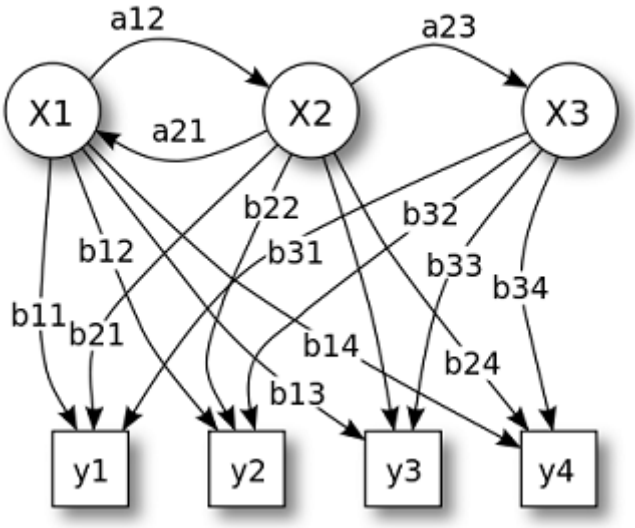


Slika 3.8: Primer markovske verige, kjer oznaki E in A predstavljata stanja, puščice pa prikazuje verjetnost ohranitev stanja ali prehoda v drugo stanje.

Model izhaja iz markovske lastnosti, ki pravi, da je mogoče razvoj markovskega procesa napovedati le na osnovi sedanjega stanja, ne da bi se sklicevali na zgodovino. Torej je prihodnje stanje neodvisno od preteklega, kar prikazuje spodnja slika (3.9).



Slika 3.9: markovska lastnost



x – stanja

y – možni izhodi (opazovana)

a – verjetnostni prehodi med stanji

b – izhodne verjetnosti

Slika 3.10: Primer skrite markovske verige.

Skriti markovski model, prikazan na sliki 3.10, ni nič drugega kot končni stohastični avtomat. Sestavljen je iz končnega števila stanj $X = x_0, x_1, x_2, ..., x_n$, kjer je x_0 začetno stanje. Stanja v oblikoslovnem označevanju predstavljajo oznake besed. Znane so samo verjetnosti prehodov $(a_{12}, a_{23}, a_{21}, ...)$. Verjetnosti posameznih stanj, neupoštevajoč verjetnosti prejšnjih stanj, izračunamo z algoritmom forward[9]. Najbolj verjetno zaporedje stanj do opazovane spremenljivke y izračuna Viterbi algoritem[10]. Za glajenje, to je računanje robnih verjetnosti vseh skritih stanj, glede na zaporedje opazovanih spremmljivk, uporabljamo Forward-backward[11] algoritem. Primer uporabe nadzorovanega HMM prikazuje slika 3.11.

```
46  import nltk
47
48  # Pridobivanje vseh oznak iz korpusa. Vsaka oznaka predstavlja
49  # stanje (state) v HMM.
50  states = list(set([tag for sent in words_tag for (word,tag) in sent]))
51
52  # Pridobivanje vseh besed v korpusu. Vsaka beseda predstavlja
53  # simbol (symbol) v HMM.
54  symbols = list(set([word for sent in words_tag for (word,tag) in sent]))
55
56  # Primer uporabe različnih verjetnostnih metod
57  # 'Maximum Likelihood Estimation'
58  est = lambda fd, bins: nltk.MLEProbDist(fd)
59  # 'Laplace'
60  est=nltk.LaplaceProbDist
61  # 'Lindstone Estimation', kjer je vrednost gamma=0.1.
62  # Pri vrednostih gamme med 0.5 in 1 se 'Lindstone Estimation'
63  # obnaša enako kot zgornja dva.
64  def lidstone(gamma):
65      return lambda fd, bins: nltk.LidstoneProbDist(fd, gamma, bins)
66  est = lidstone(0.1)
67  # 'witten bell estimation'
68  # z njim smo dobili najboljše razultate...
69  est=nltk.WittenBellProbDist
70
71  hmm_train = nltk.HiddenMarkovModelTrainer(states,symbols)
72  hmm = hmm_train.train_supervised(train_sents,est)
73
74  # Evaluacija označevalnika
75  print '%.2f%%' % (100 * hmm.evaluate(test_sents))
```

Slika 3.11: Označevanje z skritim markovskim modelom.

3.1.5 Uporaba orodja Obeliks

Program Obeliks je bil izdelan v okviru projekta sporazumevanje v slovenskem jeziku[3]. Označevalnik je sestavljen iz treh komponent: tokenizacijskega modula, ki za stavčno segmentacijo uporablja pravila, oblikoskladenjskega označevalnika ter lematizatorja LemmaGen[5], ki je prilagojen delovanju v kombinaciji z označevalnikom. Učno množico pri tem označevalniku predstavlja korpus ssj500k[13], ki je označen po tabeli oznak JOS[23]. Pri sistemu JOS s 1.903 možnimi oznakami je natančnost označevalnika 91,34 % za celotno oznako in 98,30 % za vrhnjo kategorijo (POS). Natančnost lematizacije je 97,88 % ob upoštevanju velike začetnice ter 98,55 % na ravni črkovnega niza. Označevalnik je prosto dostopen na[24]. Bolj natančneje opisano v članku[12].

Lematizacija

Naprej bomo na kratko opisali lematizaciji soroden postopek krnjenje (ang. "stemming"). Originalno ga je za angleščino napisal Martin Porter, zato ga poimenujemo „Porter Stemming algorithm". Glavna razlika med krnjenem in lematizacijo je v tem, da krnjenje besede ne pretvori v slovarsko obliko, ampak preprosto odreže končnico besede tako, da ostane le njen koren, ki pa ni nujno obstoječa knjižna beseda. Pri krnjenju se lahko več besed zlije v isti koren in tako pride do izgube informacije. Lahko bi ga opisali kot delna lematizacija. Uporabljamo ga kot približno rešitev tam, kjer si lahko privoščimo manj natančne preslikave besednih zvez v njihove korene.

Lematizacija[4] ali geslenje je postopek določanja lem oziroma osnovnih slovarskih oblik besedam. Lema besede ali besedne enote je njena osnovna morfološka različica iz slovarja. Lematizacija je zahtevnejša od krnjenja in je za večino aplikacij primernejša. Uporabljamo jo, ko nas zanima samo besedni pomen, ne pa tudi dodatne informacije, kot so sklanjatve ali slovnični časi. Program za lematizacijo besed, imenujemo lematizator (angl. Lemmatizer). Uporabljali smo lematizator narejen v okviru projekta LemmaGen[5], vgrajen v programu Obeliks[12].

Na sliki 3.11 je demonstrirana uporaba metod ovijalnega modula obeliks.py, ki prikazuje uporabo programa Obeliks v Python okolju. Prikazane so samo nekatere metode, ostale najdemo v izvorni kodi metode demo() v modulu \Diploma\obeliks.py, kjer so bolj podrobno opisani posamezni parametri.

```
80  # Za potrebe vseh naslednjih operacij najprej kreiramo objekt obeliks
81  _obeliks = obeliks()
82
83  # Neoznačeno besedilo
84  input = os.path.join(_obeliks.obeliks_path, 'examples\\vhod.txt')
85  # Ime datoteke označenega besedila
86  output = os.path.join(_obeliks.obeliks_path, 'examples\\izhod.xml')
87
88  # Označevanje vhodnega besedila
89  _obeliks.pos_tag(input, output, True, True, True)
90
91  # Učenje oblikoskladenjskega označevalnika
92  output_model = os.path.join(_obeliks.obeliks_path,
93      'examples\\POS_Model_test.bin')
94  _obeliks.pos_tag_train(output, output_model, True, 2, 10, 2, True)
95
96  # Učenje lematizatorja
97  output_model = os.path.join(_obeliks.obeliks_path,
98      'examples\\LemmatizationModel_test.bin')
99  _obeliks.lemmatizer_train(output, output_model, True, True, True, True)
100
101 # Lematizacija datoteke
102 input = os.path.join(_obeliks.obeliks_path, 'examples\\NotLemmatized.txt')
103 output = os.path.join(_obeliks.obeliks_path, 'examples\\Lemmatized.txt')
104 _obeliks.lemmatize(input, output)
```

Slika 3.12: Primer uporabe programa Obeliks.

3.2 Serializacija

Učenje modelov za določanje oblikoslovnih označb je lahko časovno zelo zahteven proces in da bi se izognili ponavljanju tega, smo uporabili serializacijo. Serializacija je pretvorba objekta v tok zaporednih podatkovnih enot za ka-

snejšo uporabo. Obratni postopek pretvorbe toka podatkov v objekte imenujemo deserializacija. V Pythonu se ta postopek imenuje pickling oziroma unpickling. Za potrebe diplomske naloge smo uporabili optimiziran modul napisan v jeziku C, ki je lahko kar 1000 hitrejši od Python različice. Primer prikazuje slika 3.13.

```
109  from cPickle import dumb
110  from cPickle import load
111
112  def savePickle(model,modelName):
113      pathPicle = '..Diploma\\sloveneTaggers'
114      fullNamePathPickle = pathPicle+modelName+'.pickle'
115      f = open(fullNamePathPickle, 'wb')
116      #Uporaba protokola vpliva na velikost shranjene datoteke.Parameter
117      #protocol ima lahko naslednje vrednosti:
118      # - '0' je originalni ASCII protokol,ki je nazaj združljiv s
119      #starejšimi različicami Pythona
120      # - '1' je starejši binarni format združljiv s starejšimi
121      #različicami Pythona
122      # - '2' ali HIGHEST_PROTOCOL je bil uveden v Pyhtonu 2.3 in
123      # zagotavlja veliko bolj učinkovito (de)serializacijo objektov
124      cPickle.dump(model, f,protocol=cPickle.HIGHEST_PROTOCOL)
125      f.close()
126
127  def loadPickle(picklePath):
128      file = open(picklePath, 'rb')
129      model = cPickle.load(file)
130      return model
131
132  # primer uporabe (de)serializacije z Brillovim modelom
133  model=NgramBrillTagger(train_sents,test_sents)
134  # serializacija objekta
135  savePickle(model,'BrillTagger')
136  # deserializacija objekta
137  model=loadPickle('..Diploma\\sloveneTaggers\\BrillTagger.pickle')
```

Slika 3.13: Primer serializacije iz objektov.

3.3 Rezultati oblikoslovnih označevalnikov

V tem poglavju ovrednotimo označevalne algoritme. Za teste smo uporabili podatke iz korpusa ccGigafida. Zaradi velikosti korpusa smo naključno izbrani segment razdelili na sedem različno velikih delov, nad katerimi smo uporabili algoritme za strojno učenje. Za lepši prikaz grafov na slikah 3.14 in 3.15 smo število besed uporabljenih pri učenju modelov normalizirali z desetiškim logaritmom. Zanimala sta nas točnost označevanja (tabela 3.3) in hitrost učenja algoritmov (tabela 3.4). Pri večjih testnih besedilih je učenje skritega markovskega modela (ang. okrajšava, HMM) in maksimalne entropije trajalo predolgo, zato smo jih predčasno ustavili. Manjkajoče vrednosti smo označili z znakom "?". Točnost označevalnih algoritmov smo računali z enačbo $T = \frac{N_p}{N} * 100\%$, kjer je N_p število pravilno označenih besed in N število vseh besed testnega besedila. Pri analizi točnosti na manjšem številu besed sta se klasifikacijska modela naivni Bayes in maksimalna entropija izkazala veliko bolje kot ostali. To je posledica uporabe značilk in s tem lažja prilagoditev učnim podatkom.

log_{10}(število besed)	2.8	3.9	4.6	5.0	5.6	6.0	6.4
število besed	611	6.999	42.607	105.168	403.163	936.666	2.576.282
Trigram	32%	50%	58%	68%	78%	83%	85%
Brill	33%	51%	59%	69%	80%	85%	88%
HMM	35%	53%	63%	75%	85%	?	?
Naivni Bayes	45%	67%	73%	80%	85%	87%	89%
Maksimalna Ent.	47%	68%	73%	?	?	?	?

Opomba: "?"-manjkajoče vrednosti zaradi časovno prepotratnega učenja.

Tabela 3.3: Prikaz natančnosti označevanja označevalnikov naučenih na različno velikih delih korpusa ccGigafida.

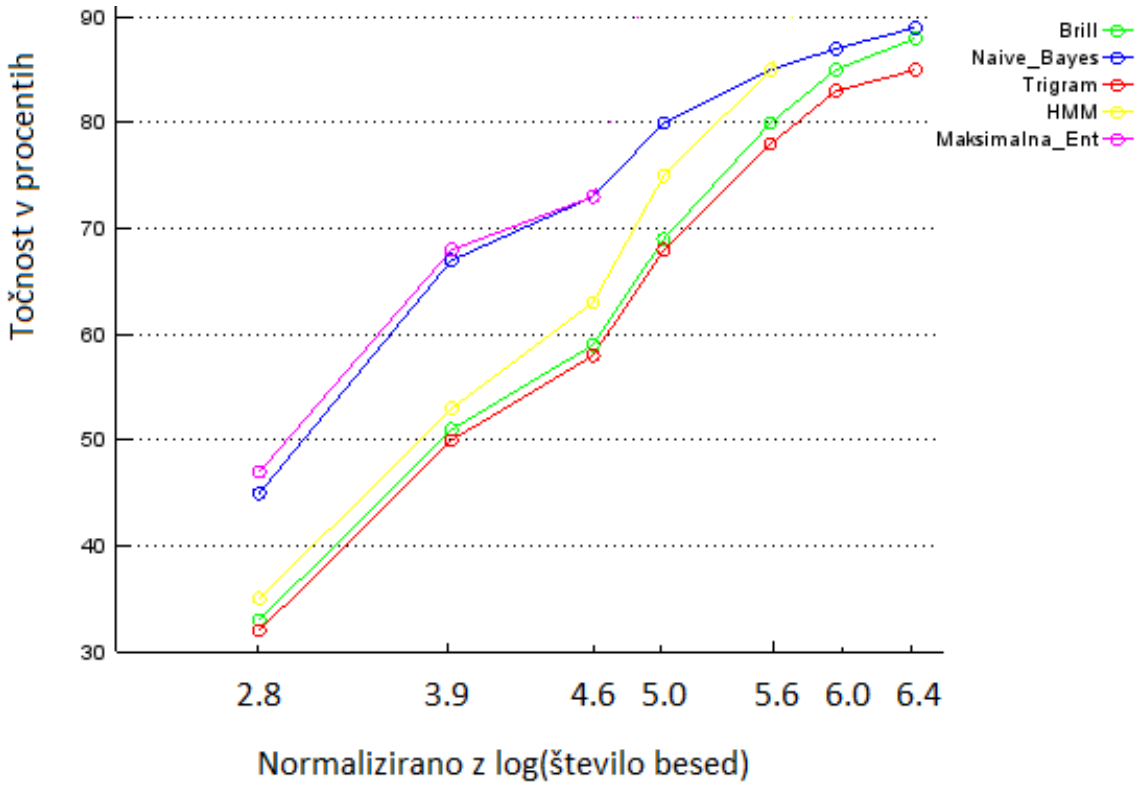
Omeniti je potrebno, da je bila pri učenju maksimalne entropije uporabljena počasnejša optimacijska metoda, vendar nas je vseeno presenetila

počasnost učenja. Pri zadnjem besedilu z 2.576.282 besed je učenje trigrama daleč najhitrejši. Izstopa čas potreben za učenje Brillovega modela na besedilu z 403.163 številom besed (638,86 sekund). Za več kot dvakrat večje besedilo z 936.666 besed smo potrebovali le 229,29 sekund. Test smo večkrat ponovili, vendar s podobnim končnim izzidom.

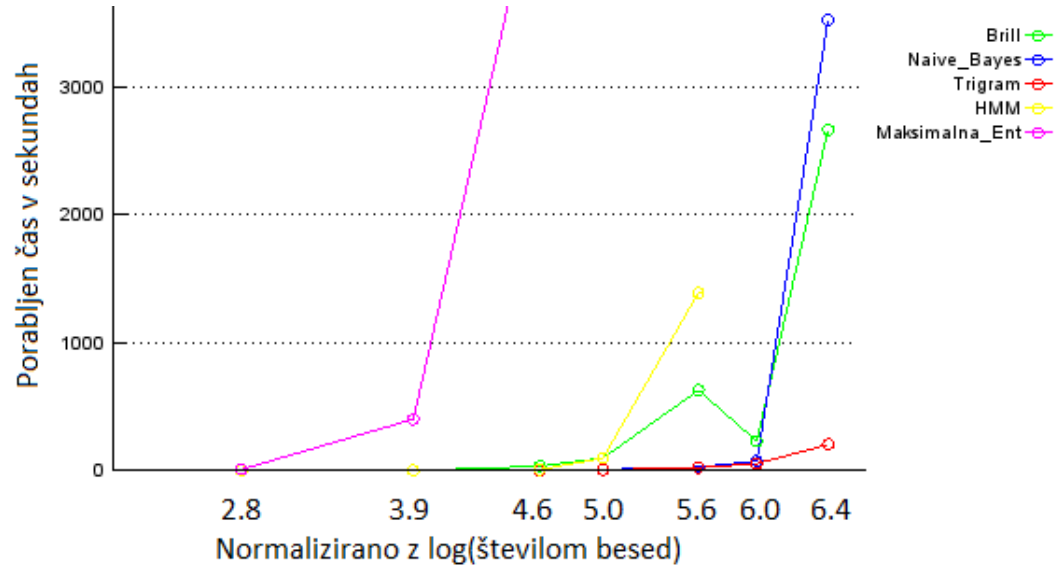
log_{10}(število besed)	2.8	3.9	4.6	5.0	5.6	6.0	6.4
število besed	611	6.999	42.607	105.168	403.163	936.666	2.576.282
Trigram	0,03	0,36	2,28	6,03	23,63	54,20	203,92
Brill	0,04	0,45	34,37	97,87	628,86	229,29	2667,42
HMM	0,01	0,29	8,3	97,35	1390,66	?	?
Naivni Bayes	0,06	0,47	2,81	7,59	27,39	73,72	3523,14
Maksimalna Ent.	9,62	402,27	4723,28	?	?	?	?

Opomba: "?"-manjkajoče vrednosti zaradi časovno prepotratnega učenja.

Tabela 3.4: Porabljeni čas v sekundah za učenje označevalnikov na različno velikih delih korpusa ccGigafida.



Slika 3.14: Prikaz točnosti oblikoslovnega označevanja modelov.



Slika 3.15: Prikaz hitrosti učenja označevalnikov.

Poglavje 4

Razčlenjevanje besedila

Skladenjski razčlenjevalnik[14] je program, s pomočjo katerega besedam v povedih pripišemo skladenjska razmerja. Z jezikoslovnega vidika avtomatizirano skladenjsko razčlenjevanje na ogromnih besedilnih korpusih omogoča raziskavo temeljnih skladenjskih pojavov za slovenščino v dejanski rabi. Skladenjsko razčlenjevanje na področju jezikovnih tehnologij predstavlja enega od vmesnih korakov analize besedila do kompleksnejših jezikovnih nalog, kot so strojno prevajanje, luščenje informacij (ang. NER ali Named Entity Recognition)[1] (poglavje 5), odgovarjanje na vprašanja, govorno komuniciranje itd. Za opis strukture povedi se pogosto uporabljajo skladenjska drevesa (Abstract syntax tree - AST) prikazano na sliki 4.1 in 4.2.

Pri podatkovno orientiranem razčlenjevanju je vir učnih in testnih podatkov drevesnica, to je besedilni korpus, ki je ročno označen z drevesi odvisnosti. Točnost razčlenjevanja ocenimo tako, da avtomatsko zgrajena drevesa odvisnosti iz besedila v testnem delu drevesnice primerjamo z ročno zgrajenimi drevesi.

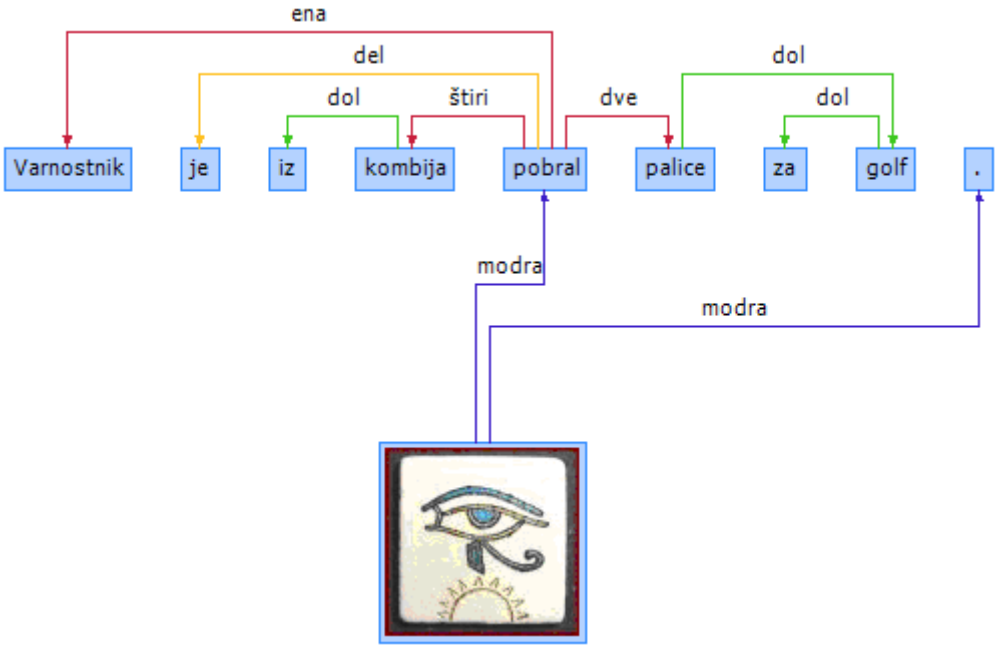
[1]NER je proces, ki zaporedjem besed določi označbe. To so lahko imena oseb in podjetji ali imena genom in proteinom

```
<p xml:id="0">
  <s xml:id="0.0">
    <w lemma="varnostnik" msd="Somei" xml:id="0.0.1">Varnostnik</w>
    <S />
    <w lemma="biti" msd="Gp-ste-n" xml:id="0.0.2">je</w>
    <S />
    <w lemma="iz" msd="Dr" xml:id="0.0.3">iz</w>
    <S />
    <w lemma="kombi" msd="Somer" xml:id="0.0.4">kombija</w>
    <S />
    <w lemma="pobrati" msd="Ggdd-em" xml:id="0.0.5">pobral</w>
    <S />
    <w lemma="palica" msd="Sozmt" xml:id="0.0.6">palice</w>
    <S />
    <w lemma="za" msd="Dt" xml:id="0.0.7">za</w>
    <S />
    <w lemma="golf" msd="Sometn" xml:id="0.0.8">golf</w>
    <c xml:id="0.0.9">.</c>
    <links>
      <link afun="ena" dep="0.0.1" from="0.0.5" />
      <link afun="del" dep="0.0.2" from="0.0.5" />
      <link afun="dol" dep="0.0.3" from="0.0.4" />
      <link afun="štiri" dep="0.0.4" from="0.0.5" />
      <link afun="modra" dep="0.0.5" from="0.0.0" />
      <link afun="dve" dep="0.0.6" from="0.0.5" />
      <link afun="dol" dep="0.0.7" from="0.0.8" />
      <link afun="dol" dep="0.0.8" from="0.0.6" />
      <link afun="modra" dep="0.0.9" from="0.0.0" />
    </links>
  </s>
</p>
```

Slika 4.1: Primer skladenjsko razčlenjene povedi v formatu XML-TEI.

Skupina povezav	Tip povezave	Kaj povezuje
Povezave prvega nivoja označujejo razmerja znotraj besednih zvez.	dol	Jedro in določilo besednih zvez.
	del	Deli zloženega povedka.
	prir	Jedra v prirednih zvezah znotraj stavka.
	vez	Besede ali ločila v vezniški vlogi.
	skup	Nepolnopomenske besede,ki imajo zelo močno tendenco po sopojavljanju.
Povezave drugega nivoja označujejo stavčne člene.	ena	Osebek stavka.
	dve	Predmet stavka.
	tri	Prislovno določilo lastnosti.
	štiri	Ostala prislovna določila.
Povezava tretjega nivoja se uporablja za povezovanje vseh ostalih struktur.	modra	Hierarhično najvišje pojavnice, skladenjsko manj predvidljive in oddaljene strukture, vrinki, ločila.

Tabela 4.1: Razlaga oznak skladenjsko razčlenjene povedi iz slike 4.1.



Slika 4.2: Primer vizualizacije skladenjsko razčlenjene povedi iz slike 4.1. Vizualizacija je zgajena na osnovi orodja dependecy parser v obliki spletnega servisa, dostopno na[29].

4.1 Skladenjski razčlenjevalnik za slovenščino

Skladenjski razčlenjevalnik za slovenščino (dependency parser) je zasnovan na razčlenjevalniku MSTParser (Minimum Spanning Tree Parser) in je bil izdelan v okviru projekta Sporazumevanje v slovenskem jeziku[3]. Naučen je na učnem korpusu ssj500k[13], ki vsebuje 11.411 ročno preverjenih povedi, razčlenjenih po sistemu odvisnostne drevesnice JOS. Pri sistemu JOS je natančnost razčlenjevalnika 90,43 % za napovedane povezave in 87,52 % za napovedane in označene povezave. Razčlenjevalnik je prosto dostopen pod licenco Apache license V2.0 na[30]. Več najdeno v[15]. Primer uporabe je prikazan na sliki 4.3.

```
143  # Kreiramo objekt dependecyParser().
144  _parser = dependecyParser()
145
146  # Primer razčlenjevanja
147  input = 'test\\tagged_test.xml'
148  output = 'test\\parsed_test.xml'
149  model = 'small'
150  _parser.parse(input, model, output, False)
151
152  # Primer učenja razčlenjevanja
153  input_parsed = 'test\\jos5.xml'
154  output_model = 'test\\test.model'
155  # Parameter z vrednostjo '15' označuje število iteracij učenja.
156  # Privzeta vrednost je nastavljena na 10.
157  _parser.train(input_parsed, output_model, 15, True)
158
159  # Primer evaluacije naučenega modela
160  results = 'eval\\eval_result.txt'
161  # Parameter z vrednostjo '3' označuje uporabo
162  # 3-kratnega prečnega preverjanja.
163  # Parameter z vrednostjo '15' označuje število iteracij učenja.
164  _parser.evaluate(input_parsed, results, 3, 15, True)
```

Slika 4.3: Primer skladenjskega razčlenjevanja besedila.

Poglavje 5

Razpoznavanje entitet

Z določanjem pomena besed v besedilu se ukvarjajo metode razpoznavanja entitet (ang. Named Entity Recognition ali krajše NER). Običajno so modeli naučeni za določanje vnaprej opredeljenih kategorij, kot so imena krajev, oseb, organizacij,... Alternativa temu je uporaba predznanja v obliki leksikonov. Njihova slabost je, da ne zaznajo neznanih entitet, če jih nimajo v obstoječem leksikonu. Zato jih pogosto kombiniramo s sistemom, zasnovanem na strojnem učenju, tako da tvorita hibriden sistem. Eden izmed takih sistemov je SLNER, ki smo ga uporabili v naši nalogi.

Uporabo NER običajno najdemo pri pridobivanju informacij (poglavje 6), v bioinformatiki, identifikaciji oseb, v molekularni biologiji, sistemih za odgovarjanje na vprašanja (question-answering system), spletnem iskanju itd. Prepoznavanje entitet ni enostavna naloga. Predvsem pride do nejasnosti pri odkrivaju oseb in organizacij. Lahko se zgodi, da ima neka entiteta več pomenov. Primer so recimo priimki: po poklicu (Kovač, Tkalec), po živali (Medved, Ježek), po prebivalcu države (Čeh, Nemec), Kralj, Božič, Petek, Koren, Oblak,... Prav tako je z imeni podjetij, ki so pogosto izbrana po ustanoviteljih (Maks Horvat s.p.).

5.1 Orodje SLNER

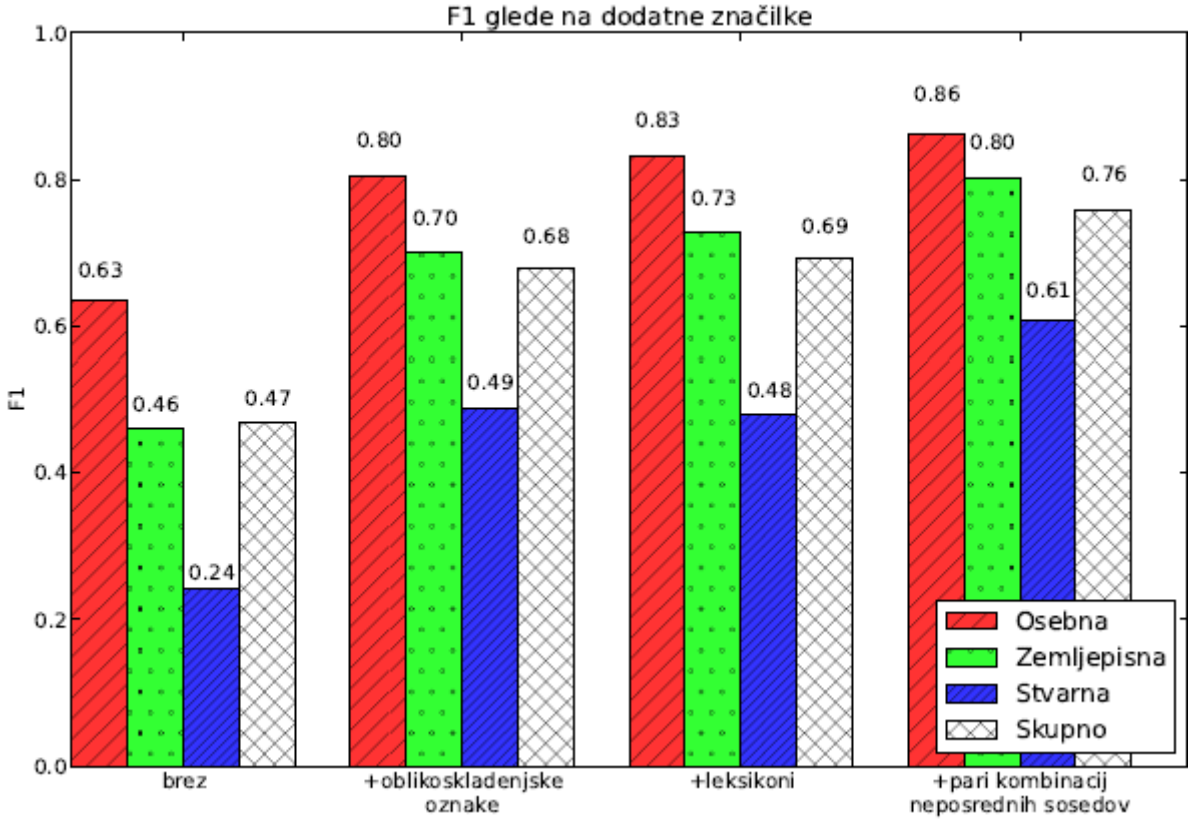
Orodje uporablja nadzorovano učenje s pogojnimi naključnimi polji (Conditional Random Fields, ali krajše CRF), ki temelji na javi napisani knjižici Mallet (McCallum, 2002)[16]. Entitete, ki jih program zaznava so lastna, zemljepisna in stvarna imena. Naučen je na označenem korpusu ssj500k[13]. V kombinaciji z ostalimi značilkami doseže sistem na testni množici 77 % natančnost in 75 % priklic, pri čemer so lastna in zemljepisna imena razpoznana bistveno bolje kot stvarna, saj je razred stvarnih imen zelo raznolik in zato težaven za učenje. Rezultati so prikazani v tabeli 5.1 in 5.2 ter na sliki 5.1. Program je prosto dostopen pod licenco Apache 2.0 na[31]. Bolj obširno napisano v članku[17].

Tip entitete	**Natančnost**	**Priklic**	F_1
Brez oblikoskladenjskih oznak			
Osebna	0.6207	0.6533	0.6342
Zemljepisna	0.4426	0.4868	0.4595
Stvarna	0.3171	0.1970	0.2412
Skupno	0.4932	0.4464	**0.4681**
Z oblikoskladenjskimi oznakami			
Osebna	0.7632	0.8526	0.8046
Zemljepisna	0.7303	0.6770	0.7016
Stvarna	0.5756	0.4283	0.4881
Skupno	0.7011	0.6585	**0.6788**

Tabela 5.1: Rezultati poskusov glede na uporabljene oblikoskladenjske oznake

Tip entitete	**Natančnost**	**Priklic**	F_1
Z uporabo leksikonov			
Osebna	0.7851.	0.8832	0.8307
Zemljepisna	0.7727	0.6892	0.7280
Stvarna	0.5524	0.4261	0.4797
Skupno	0.7126	0.6718	**0.6914**

Tabela 5.2: Rezultati poskusov glede na uporabljene leksikone



Slika 5.1: F1 glede na najboljše kumulativno dodane značilke.

Primer uporabe orodja SLNER je na sliki 5.2.

```
1  # kreiranje objekta slner
2  _slner = slner()
3
4  # Primer vhodne datoteke predhodno označene z
5  # oblikoslovnimi oznakami(glej poglavje 3)
6  input = os.path.join(_slner.slner_path,
7      'examples\\toBeTagged.xml')
8  # Ime izhodne datoteke
9  output = os.path.join(_slner.slner_path,
10     'examples\\corpus_with_entities.tei.xml')
11
12 # Označevanje NER
13 _slner.tag(input, output)
14
15 # Ime še neobsoječega naučenega modela
16 output_model = os.path.join(_slner.slner_path,
17     'examples\\Test.ser.gz')
18 # Učenje NER modela
19 _slner.train(input, output_model)
20
21 # primer evaluacije naučenega modela
22 _slner.evaluate(input)
```

Slika 5.2: Primer uporabe orodja za razpoznavanje imenskih entitet.

Poglavje 6

Poizvedbe po informacijah

Poizvedovanje po informacijah je eden od pripravljalnih korakov pri tekstovnem rudarjenju. Poskuša zagotoviti informacije, o katerih bi uporabnik želel izvedeti več. Začne z vnosom uporabnikove poizvedbe. Naloga sistem je, da iz nestrutkuriranih dokumentov razbere čim več ustreznih informacij in jih vrne. Stopnjo ustreznosti smo računali z naslednjimi algoritmi: obrnjen index (inverted index), logični model (boolean retrieval), vektorski model (tf-idf), kosinusna podobnost (cosine similarity) . Poizvedba je preslikava med uporabnikovim vprašanjem in iskano informacijo. Zaradi hitrosti izvajanja se iskanje ne izvaja nad dejanskim besedilom, ampak nad vnaprej pripravljenim katalogom, pripravo katerega imenujemo indeksiranje (ang. text indexing). Najstarejša računalniško podprta oblika tekstovne zbirke je bibliografska zbirka. V njej niso shranjeni dokumenti, ampak le njihovi nadomestki, ki jim pravimo bibliografski zapisi. Bibliografski zapis je sestavljen iz polj, ki vsebujejo tiste podatke o dokumentu, ki so potrebni za njegovo nedvoumno identifikacijo: ime avtorja, naslov dokumenta, datum nastanka,...

Pri iskanju po relacijski zbirki, so kriteriji selekcije podatkov znani. Iskanje je strukturirano. Pri sintaktično pravilni poizvedbi nam sistem vrne vse iskane vrednosti. Iskanje v relacijski zbirki je deterministično. V tekstovnih zbirkah večinoma poizvedujemo po vsebini dokumentov. Vsebina je komple-

ksna lastnost dokumenta, ki se je ne da izraziti z enostavnimi vrednostmi. Relevanten dokument bomo našli v tekstovni zbirki le z neko verjetnostjo. Pravimo, da je iskanje v tekstovni zbirki verjetnostno.

Implementirali smo "ad-hoc"[18] rešitev. Po prvem zagonu modul ..Diploma\IRSystem.py ustvari imenik z imenom lemmatized v ..Diploma\InformationRetrieval\data\SloCorpus. To je predpomnilnik za neobdelane dokumente v imeniku ..Diploma\InformationRetrieval\data\SloCorpus\raw. Kasnejši zagoni bodo veliko hitrejši, saj lematizacija ne bo več potrebna. V primeru dodajanja ali sprememb besedil se direktorij lemmatized izbriše in modul ponovno zažene. Na sliki 6.1 pripravimo besedila za testiranje algoritmov.

```
170  # Inicializiramo spremenljivke
171  irsys = IRSystem()
172
173  # Beremo vsa besedila v direktoriju raw. Postopek
174  # vključuje tokenizacijo in lematizacijo besedil. To
175  # storimo samo prvič. Po končanem procesu dobimo direktorij
176  # lemamatized, v katerem so vsa besedila lematizirana.
177  irsys.read_data(sys.path[0]+'..InformationRetrieval\\data\\SloCorpus')
178
179  # indeksiranje dokumentov
180  irsys.index()
181
182  # računanje tf-idf, potrebujemo za uporabo TF-IDF in
183  # kosinusne podobnosti
184  irsys.compute_tfidf()
```

Slika 6.1: Priprave za testiranje algoritmov opisanih v naslednjih podpoglavjih.

6.1 Obrnjeni indeks

Obrnjeni indeks (inverted index[1]) je indeks podatkovne strukture, ki za vsako besedo hrani seznam, v katerih dokumentih se pojavi, kolikokrat in na katerem mestu. Namen obrnjenega indeksa je hitro in celovito iskanje po dokumentih na račun povečanega procesiranja, ko dodajamo nov dokument. Primer uporabe je prikazan na sliki 6.2.

```
1  # Poizvedba
2  query = "Bajtarica"
3  # Poizvedba po dokumentih
4  doc_retrieve = irsys.get_posting_unlemmatized(query)
5
6  #Izpis
7  print doc_retrieve
```

Slika 6.2: Primer uporabe obrnjenega indeksa.

6.2 Preiskovanje z logičnimi izrazi

Zgodnji sistemi za iskanje informacij so uporabnikom omogočali oblikovanj povprašanj z uporabo logičnih operatorjev (IN, ALI, NE). Model se je uveljavil pred dobrimi 30 leti, ko so bile na voljo skromne računalniške kapacitete. Bistveni značilnosti logičnega modela sta natančno definiran odnos med posameznimi členi iskalne zahteve in razvrstitev zapisov v zbirki v jasno omejeni množici relevantnih in nerelevantnih zapisov. Primer uporabe je prikazan na sliki 6.3.

[1] imenovan tudi postings file ali inverted file

```
1  # Poizvedba
2  query = "v gozd je šel"
3  # Poizvedovanje po dokumentih
4  doc_retrieve = irsys.query_retrieve(query)
5
6  # Izpis vseh skupnih dokumentov besedam
7  # v poizvedbi
8  print doc_retrieve
```

Slika 6.3: Primer logične poizvedbe.

6.3 Vektorski model

Logični model je uporaben za uporabnike z znanjem Boolove algebre, ki točno vedo, katere informacije iščejo. Večini uporabnikom ta način ne ustreza, saj bodisi ne znajo pisati logičnih poizvedb ali pa se jim preprosto ne ljubi. Še ena slabost takih poizvedb je, da prikažejo premalo („in“ operator) ali preveliko („ali“ operator) število rezultatov. Potrebno je veliko znanja, da se sestavi poizvedba z obvladljivim številom vrnjenih rezultatov.

Za računanje uteži se uporablja TF-IDF (term frequency-inverse document frequency) shema. Sodi v rangirana iskanja, ki vrnejo rezultat glede na pomembnost besede v dokumentu. Namesto logičnih operatorjev ali izrazov je uporabnikova poizvedba niz ene ali več besede v naravnem jeziku. Za razliko od logičnega modela število vrnjenih rezultatov ni težava. Lahko se odločimo, da izberemo 10 najboljših rezultatov in tako uporabnika ne preplavimo z nepotrebnimi informacijami.

TF-IDF določi utež, ki je:

- višja, če se beseda večkrat pojavi v majhnem številu dokumentov,
- manjša, če se beseda pojavi malokrat v dokumentu ali se pogosto pojavlja v številnih dokumentih,
- najnižja, če se beseda pojavi v večini ali vseh dokumentih (običajno so

to mašila[2] ali pogoste besedne zveze.)

Primer poizvedbe z TF-IDF uteževanjem je prikazan na sliki 6.4.

```
1  # Poizvedba
2  query = "Bajtarica, 0"
3  # Poizvedovanje po dokumentih
4  doc_retrieve = irsys.get_tfidf_unlemmatized(word)
5
6  # Izpis
7  print doc_retrieve
8
```

Slika 6.4: Primer poizvedbe z TF-IDF uteževanjem.

6.4 Kosinusna podobnost

TF-IDF rangira rezultate iskanja po pomembnosti besed v dokumentih, vendar ima slabost, da poizvedbe obravnava kot vrečo besed (bag of words model)[3]. Algoritem opisan v tem poglavju odpravi to slabost.

Kosinusna podobnost opisuje relativni pomen izrazov v dokumentu. Če si predstavljamo koordinatni sistem, besede predstavljajo osi, dokumenti in poizvedbe pa vektorje v skupnem vektorskem prostoru. Podobnost računamo kot kot med vektorjevima vrednostima v n-dimenzionalnem prostoru (n je število vrednosti v vektorju).

$$\cos(\overrightarrow{q}, \overrightarrow{d}) = \frac{\overrightarrow{q} \bullet \overrightarrow{d}}{\mid \overrightarrow{q} \mid\mid \overrightarrow{d} \mid} = \frac{\overrightarrow{q}}{\mid \overrightarrow{q} \mid} \bullet \frac{\overrightarrow{d}}{\mid \overrightarrow{d} \mid} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}} \tag{6.1}$$

[2]mašila so besede, ki jih uporabljamo za lepši prehod med povedmi in kot take ne doprinesejo nobene dodatne vrednosti in se jih skladno s tem odstrani iz besedila, tako zmanjšamo velikost korpusa, hitrost učenja,... Načeloma so to besede: in, ali, ter, z, s, ko, temveč,...

[3]V tem modelu so besedila predstavljeno kot neurejena zbirka besed brez upoštevanega vrstnega reda. To pomeni, da je stavek: „Pišem diplomo doma pred računalnikom.“ verjetnostno enakovreden „pred Pišem doma diplomo računalnikom.“

q_i - TF-IDF utež i-te besede v poizvedbi
d_i - TF-IDF utež i-te besede v dokumentu
$|V|$ - vektorski prostor
$cos(\overrightarrow{q}, \overrightarrow{d})$ - kot med $\overrightarrow{q}$ in $\overrightarrow{d}$, vrednosti so lahko med 0(ponavadi pomeni neodvisnost) in 1(kar pomeni natanko enaka)

Primer poizvedbe z kosinusno podobnostjo je prikazan na sliki 6.5.

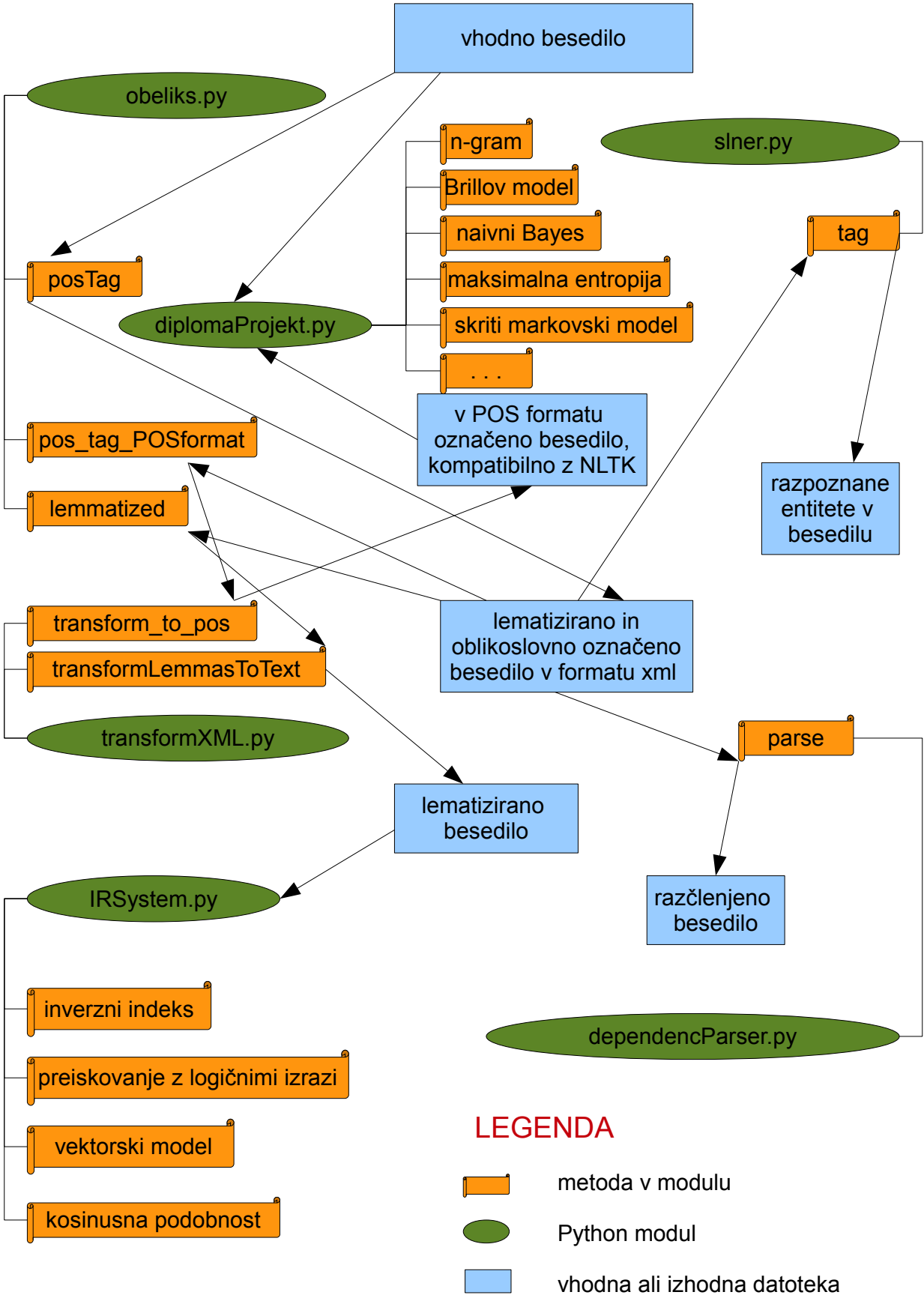
```
1  # Poizvedba
2  query = "v gozd je šel"
3  # Poizvedovanje po dokumentih
4  doc_retrieve = irsys.query_rank(query)
5
6  # Izpis
7  print doc_retrieve
8
9
```

Slika 6.5: Primer poizvedbe z kosinusno podobnostjo.

6.5 Zasnova sistema

Interakcijo med posameznimi moduli smo prikazali na sliki 6.6. Na sliki nismo navedli vseh metod programa, le tiste, ki so potrebne za povezavo med posameznimi moduli. Program je zasnovan tako, da omogoča vključitev novih orodij za obdelavo naravnega jezika. V primerih, da je orodje napisano v drugem jeziku kot Python in omogoča delovanje preko ukazne vrstice, dodamo nov Python modul s potrebnimi metodami. Poskrbeti moramo za formate izhodnih in vhodnih datotek kompatibilne s knjižnico NLTK. Pri modulu dependencyParser.py bi bilo potrebno izhodne datoteke razčlenjenega besedila pretvoriti v NLTK format imenovan "IOB tags". V tem formatu je vsaka beseda oziroma sklop zaporednih besed označen z oznakami: "I" (inside), "O" (outside), "B" (begin). Beseda je označena z oznako "I", če označuje začetek bloka povedi, "I" označuje vmesno sekvenco bloka, oznaka

"O" pa zunanji del bloka. NLTK poleg "IOB" oznak ponuja še predstavitev z drevesno strukturo.



Slika 6.6: Shema uporabe posameznih modulov

Poglavje 7

Sklepne ugotovitve

V diplomski nalogi smo reševali problem avtomatskega določanja besednih vrst za slovenski jezik. Uporabili smo algoritme v knjižnici NLTK. Iz korpusa Gigafida smo zgradili označevalnike n-gram, Brill, Naive Bayes, maksimalna entropija in skriti markovski model. V NLTK smo vključili orodje Obeliks za lematizacijo in oblikoslovno označevanje besedil. Za razčlenjevanje besedil in razpoznavanje imenskih entitet smo uporabili orodji dependecy parser in slner. Razvili smo orodje za poizvedovanje po informacijah in ga testirali na skupini dokumentov. Pri tem smo uporabljali: obrnjeni indeks, logične izraze, vektorski model in kosinusno podobnost. Razvili smo črkovalnik z možnostjo popravljanja napačnih besed, ki deluje na "noisy channel model", vendar nam je zmanjkalo časa za primerno vključitev v NLTK. V diplomskem delu smo opise metod in algoritmov podprli s praktičnimi primeri.

Motiv za nadaljne delo bi lahko bila boljša povezanost vseh orodij z izborom skupne tehnologije ali enega jezika. Trenutno delo poteka preko programskih ukazov oz. klicanja metod, kar bi se dalo izboljšati z izdelavo uporabniškega vmesnika. Implementirali bi funkcionalnosti za druge namene v okviru NLP: jezikovni prevajalniki, klasifikacija besedil, odkrivanje plagiatorstva, grupiranje teksta,... Prostori za izboljšavo so tudi v trenutnih algoritmih. Obdelava naravnega jezika na komercialni ravni zahteva obdelavo ogromnih korpusov z mnogo značilkami, kar lahko pomeni ogro-

mno računskega časa. Smiselna bi zato bila paralelizacija opisanih metod. Razvijalci NLTK so to deloma upoštevali z uporabo Execnet[19], ki ponuja distribuirano računanje.

Literatura

[1] NLTK knjižnica. Dostopno na http://nltk.org/ (marec 2013)

[2] Steven Bird, Ewan Klein, Edward Loper, *Natural Language Processing with Python*, O'Reilly Media, 2009. Dostopno na: nltk.org/book/ (marec 2013)

[3] Projekt Sporazumevanje v slovenskem jeziku. Dostopno na: http://www.slovenscina.eu/ (2013)

[4] Matjaž Juršič. Implementacija učinkovitega sistema za gradnjo, uporabo in evaluacijo lematizatorjev tipa RDR. Diplomsko delo, Fakulteta za računalništvo in informatiko, Ljubljana, 2007.

[5] Lematizator. Dostopno na: http://lemmatise.ijs.si/ (2013)

[6] Eric Brill, A corpus-based approach to language learning. Dissertation, University of Pennsylvania, 1993

[7] Geoffrey I.Webb, Janice R. Boughton, Zhihai Wang, "Not So Naive Bayes:Aggregating One-Dependence Estimators",*Machine Learning,* vol. 58, 5-24, 2005.

[8] Wikipedia: Entropija. Dostopno na: http://sl.wikipedia.org/wiki/Entropija_(informatika) (marec 2013)

[9] Wikipedia: Forward algorithm. Dostopno na: http://en.wikipedia.org/wiki/Forward_algorithm (marec 2013)

[10] Wikipedia: Viterbi algorithm, Viterbi AJ (April 1967). "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm". IEEE Transactions on Information Theory 13 (2): 260–269. Dostopno na:
http://en.wikipedia.org/wiki/Viterbi_algorithm (marec 2013)

[11] Wikipedia: Forward backward algorithm. Dostopno na:
http://en.wikipedia.org/wiki/Forward/backward_algorithm
(marec 2013)

[12] Miha Grčar, Simon Krek, Kaja Dobrovoljc: Obeliks: statistični oblikoskladenjski označevalnik in lematizator za slovenski jezik. V T. Erjavec, J. Žganec Gros (ur.): Zbornik Osme konference Jezikovne tehnologije. Ljubljana (2012): Institut Jožef Stefan.

[13] Učni korpus ssj500k. Dostopno na:
http://www.slovenscina.eu/tehnologije/ucni-korpus (marec 2013)

[14] Domen Marinčič, Strojno razčlenjevanje besedila z iskanjem stavkov in naštevanj, doktorsko delo, 2008

[15] Kaja Dobrovoljc, Simon Krek, Jan Rupnik (2012): Skladenjski razčlenjevalnik za slovenščino. V T. Erjavec, J. Žganec Gros (ur.): Zbornik Osme konference Jezikovne tehnologije. Ljubljana: Institut Jožef Stefan.

[16] Mallet knjižnica. Dostopno na:
http://mallet.cs.umass.edu/ (marec 2013)

[17] Tadej Štajner, Tomaž Erjavec, Simon Krek (2012): Razpoznavanje imenskih entitet v slovenskem jeziku. V T. Erjavec, J. Žganec Gros (ur.): Zbornik Osme konference Jezikovne tehnologije. Ljubljana: Institut Jožef Stefan.

[18] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, Introduction to Information Retrieval, Cambridge University Press. 2008.

[19] Using Execnet for Parallel and Distributed Processing with NLTK. Dostopno na: http://www.packtpub.com/article/using-execnet-parallel-and-distributed-processing-nltk (marec 2013)

[20] Korpus ccGigafida. Dostopno na: www.slovenscina.eu/korpusi/proste-zbirke (marec 2013)

[21] Korpus Šolar. Dostopno na: www.slovenscina.eu/korpusi/solar (marec 2013)

[22] Korpus Gigafida. Dostopno na: www.slovenscina.eu/korpusi/ (marec 2013)

[23] Oblikoskladenjske specifikacije JOS. Dostopno na nl.ijs.si/jos/josMSD-sl.html (marec 2013)

[24] Orodje Obeliks. Dostopno na: http://sourceforge.net/projects/obeliks/ (marec 2013)

[25] MEGAM knjižnica. Dostopno na: http://www.umiacs.umd.edu/∼hal/megam/ (marec 2013)

[26] TADM knjižnica. Dostopno na: http://tadm.sourceforge.net/ (marec 2013)

[27] Scipy knjižnica. Dostopno na: http://www.scipy.org/ (marec 2013)

[28] Scitik-learn knjižnica. Dostopno na: http://scikit-learn.org/stable/ (marec 2013)

[29] Spletni servis razčlenjevalnika besedila. Dostopno na: Razclenjevalnik.slovenscina.eu (marec 2013)

[30] Orodje dependecy parser. Dostopno na: www.slovenscina.eu/tehnologije/razclenjevalnik (marec 2013)

[31] Orodje SLNER. Dostopno na:
http://ailab.ijs.si/~tadej/slner.zip (marec 2013)