UNIVERSITATEA "POLITEHNICA" BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE

PLN TOOLKIT PROJECT

Ingineri:
Ana Găinaru
Gabriel Sandu
Ştefan Dumitrescu

Introducere

Acest proiect a fost elaborat in cadrul cursului de Procesare a Limbajelor Naturale si foloseste ca input un fisier XML elaborat cu ajutorul programului ConcertChat, dezvoltat de catre Fraunhofer (puteti gasi mai multe informatii despre el la adresa http://www.ipsi.fraunhofer.de/concert/index_en.shtml?projects/chat) .

Una din utilizarile cele mai interesante ale ConcertChat este folosirea pentru a rezolva probleme in mod colaborativ, iar experimentele din cadrul programului VMT cu studenti au demonstrat ca problemele pot fi rezolvate mai usor in acest mod.

Proiectul PLN Toolkit isi propune sa analizeze outputul unei sesiuni tipice ConcertChat in care participantii au discutat pe mai multe subiecte, generand o discutie tipica purtata pe Internet de aproximativ 600-1000 replici.

In urma analizei acestei sesiuni de chat, se doreste obtinerea unor rezultate prin metode automate, spre deosebire de metodele de adnotare manuala deja cunoscute.

Se vor folosi algoritmi cunoscuti pentru obtinerea textului in forma tokenizata, cu ajutorul deja existentului framework NLTK pentru Python, ce contine si ontologia WordNet.

Toolkit-ul se afla disponibil sub licenta GPLv3 la URL-ul
http://project-pln.googlecode.com/

In continuare vom prezenta particularitatile fiecarui tip de analiza pe care am implementat-o si output-ul acestora.

Formatul XML ConcertChat

Am obtinut 6 fisiere XML cu exemple de sesiuni de chat, care se gasesc in directorul xml din cadrul proiectului si sunt numerotate de la 1 la 6. Formatul lor este urmatorul:

```
<Dialog team="t4">
<Participants>
<Person nickname="stefan"/>
<Person nickname="claudiu"/>
<Person nickname="ana"/>
<Person nickname="andrei"/>
<Person nickname="Andrei"/>
</Participants>
<Topics/>
<Body>
<Turn nickname="stefan">
<Utterance genid="1" ref="-1" time="18/12/2008 11:20:08">joins the
room</Utterance>
</Turn>
```

Elementul root este <Dialog> si contine <Participants>, <Topics> si <Body>. <Body> este alcatuit din <Turn> iar fiecare <Turn> contine o <Utterance> ce are

- genid: id-ul ei unic
- ref: id-ul de la o replica precedenta la care exista o legatura
- time: la ce timestamp a fost luata replica
- textul din interior

Pentru prelucrarea usoara a acestui tip de fisier, s-a scris scriptul xmlReader.py , care contine clasa XMLReader, cu metodele

- `def read(self,filename)`
- `def close(self)`

Dupa ce s-a apelat read() pentru deschiderea unui fisier, putem accesa documentul atat ca o lista de Utterances in XMLReader.doc, sau DOM tree prin XMLReader.xmldoc .

Tokenizare

Pentru tokenizare am folosit functia preferata in NLTK din pachetul nltk.tokenizer word_tokenize(text) care foloseste un algoritm mai complex decat un simplu split() pe text dupa spatii. word_tokenize foloseste un TreebankWordTokenizer care face tokenizarea prin referinta la un corpus de cuvinte, lucru ce face mai usoara detectarea sfarsiturilor de propozitie, dupa cum se poate observa din exemplul de jos:

“Mrs. Smith went to her car. \n\t She forgot the keys in the house, though”

Se poate observa ca primul punct nu este de fapt sfarsitul de propozitie, ci indica o prescurtare, lucru ce e detectat corect doar daca se foloseste un PunktWordTokenizer sau TreebankWordTokenizer.

Se elimina si white spaces: caractere tab si secventa de mai multe spatii, si sfarsitul de linie \n. Eliminarea lor se face cu un tokenizator pe baza de expresii regulate.

Eliminare Stopwords

Pentru eliminarea stopwords, s-a scris scriptul stopwords.py, care contine clasa StopWords, cu metodele:

- `def getWords1 (self, string)`
- `def getWords2 (self, string)`

Metodele primesc un string si intorc o lista de cuvinte, din care s-au scos cuvintele ce nu sunt folosite pentru extragerea radacinii (stemming). Aceste cuvinte exista in mai multe variante de liste, motiv pentru care s-au alcatuit 2 functii pentru acest proces.

Prima functie foloseste o lista de stopwords din NLTK, ce se poate gasi in nltk.corpus.stopwords.words('english'). Din pacate, aceasta lista este insuficienta deoarece in textul din chat-uri exista si o multitudine de emoticoane, ce nu au valoare sintactica pentru analize. Am alcatuit o lista orientativa de prescurtari folosite in chat si emoticoane, salvata intr-o variabila globala in acel script cu numele alt_stop_words.

Functia a doua se bazeaza pe o lista locala de stopwords. Diferenta intre liste este ca in cea din NLTK sunt trecute ca stopwords niste verbe curente, de exemplu "think", lucru ce ar face ca aceste verbe sa dispara din analiza noastra.

De aceea, am inclus doua variante ale functiei de eliminare stop words, si analizele ulterioare folosesc varianta getWords2().

Stemming

Stemming-ul reprezinta eliminarea afixelor cuvintelor flexionate. Se incearca reducerea lor la radacina, astfel putand fi comparate mult mai usor cuvintele intre ele. Exita cazuri in care aceste cuvinte nu se reduc la o radacina valida (nu se vor reduce la radacina morfologica), este suficient ca acele cuvinte inrudite sa se reduca la o radacina comuna.

Stemmerul utilizat este varianta PorterStemmer. Acesta este un stemmer de bazat pe reguli, ce incearca eliminarea afixelor iterativ.

In cadrul proiectului, are o utilizare usoara, de forma:

```
import stemmer
pstemmer = PorterStemmer()
stemmed_word = pstemmer.stem(`word`)
```

Lanturi Lexicale

Clasa ce se ocupa de gasirea lanturilor lexicale se numenste `LexChains`.

Lanturile lexicale sunt liste de cuvinte din mai multe replici ce au o legatura intre ele. Pentru proiect un lant lexical contine cuvinte ca au intre ele doua tipuri de legaturi: legaturi puternice si legaturi medii.

Legaturiile puternice intre doua cuvinte sunt reprezentare de aceleasi cuvinte, sinonime sau cuvinte cu inteles foarte apropiat (exemplu automobil, masina). Legaturile medii

sunt legaturi mai slabe intre cuvinte care au totusi un sens comun intr-un anumit context (de exemplu problema si algoritm).

Pentru a vedea gradul de similitudine dintre doua cuvinte proiectul foloseste interfata cu wordnet pusa la dispozitie de nltk. Functia folosita calculeaza cat de similare sunt doua cuvinte in functie de cea mai scurta cale in taxonomie ce conecteaza cele doua sensuri cat si inaltimea maxima in taxonomie a conceptului parinte a celor doua cuvinte. Relatia folosita este -log(p/2d) unde p este lungimea drumului cel mai scurt iar d este intltimea topologiei.

Pentru legaturi puternice se cauta valori mai mari ca 3 (sinonimele au legaturi de 3,62) iar pentru legaturi medii valori mai mari ca 1.5 (Am descoperit experimental ca acestea sunt cele mai bune valori pentru a nu exclude nici un cuvant legat de un lant lexical dar nici sa adaug cuvinte ce nu au legatura)

Algorituml de identificare este urmatorul:

Pentru fiecare replica:

- Se elimina stopwords
- Se calculeaza POST pentru cuvintele din replica.

- Se selecteaza grupurile nominale si cele verbale. Se iau in considerare si verbele deoarece si ele pot crea legaturi in cadrul replicilor. Deoarece wordnetul nu caluleaza gradul de similitate intre un verb si un adjectiv se cauta doar verbele care pot avea si forme de grup nominal (de exemplu to walk poate avea forma a walk care este un NP).
  Din acest motiv in pasul urmator se transforma toate cuvintele in forma lor de grup nominal.

- Initial se cauta legaturi intre cuvinte cu legaturi foarte puternice in toate lanturile lexicale existente in momentul respectiv. Se limiteaza cautarea in replici vechi de 10 pasi. La primul cuvant gasit care are o legatura puternica cu cuvantul in cauza cautarea se opreste si acesta este adauga in lista lexicala cu care a fost gasita legatura.

- Daca nu gasim nici un cuvant de care sa se lege puternic incepem cautarea legaturilor medii in toate lanturile lexicale si pastram pentru fiecare cuvantul cu care are legatura medie cea mai puternica. Cautarea se face pentru maxim 4 pasi in urma. La sfarsit alegem dintre toate lanturile lexicale pe acela cu legatura cea mai puternica.

- Daca nu se gaseste nici un fel de legatura se creaza un nou lant lexical ce va contine cuvantul in cauza.

Cautarea pentru legaturile puternice este relativ rapida, insa pentru legaturile medii operatia este costisitoare deoarece trebuie sa parcurgem toate lanturile lexicale de fiecare data. Pe masuta ce avem mai multe replici si numarul lanturilor creste.

Pentru cele 600 de replici timpul de rulare pentru gasirea lanturilor lexicale a fost de 7 minute, numaul lanturilor lexicale fiind de aprox 300 multe din ele insa avand un singur cuvant. Pe acestea le-am eliminat la scriere in fisier.

Scrierea in fisier se face prin adaugarea unui tag <LexChains> in interiorul tagului <Analysis> . Tagul <LexChains> contine mai multe taguri <Chain> cate unul pentru fiecare lant lexical.
Fiecare lant lexical contine la randul sau mai multe taguri <TextUt> ce contine:

- intre taguri cuvantul ce face parte din lantul lexical.
- un atribut id ce reprezinta id-ul replicii din care face parte cuvantul

(exemplu <TextUt id="578">Synset('approach.n.01')</TextUt>)

Interferenta

Clasa ce se ocupa de gasirea lanturilor lexicale se numeste `Interference`.

Gasirea interferentei intre replici porneste de la lanturile lexicale detectate anterior.

Pentru fiecare legatura puternica sau medie intre doua cuvinte ce nu fac parte din aceeasi replica am stocat intr-o lista tripletul [id replica 1, id replica 2 si gradul de similitudine dintre cele doua cuvinte din ele].
In faza urmatoare se ia fiecare replica si i se determina toate replicile anterioare care au legatura medie sau puternica care apoi se stocheaza in ordinea similitudini (replicile cu o relatie mai stransa vor fi primele in vector).

Avand pentru fiecare replica toate replicile anterioare cu care are o legatura cat si valoarea legaturii putem sa cautam tipul interferentei intre acestea.

Tipul interferentei este data de cuvintele cheie dintre replici. Lista cuvintelor “cue phrases” se gaseste in modului cuephrases si e impartita in mai multe tipuri:

- consequence
- contrast
- elaboration
- conclusion
- confirmation
- timing

In functie de categoria in care este gasit un cuvant cheie in lista de cue phrases putem trage concluzia asupra interferentei dintre replici.

Cautarea cuvintelor cheie se face la inceputul replicii a doua sau la sfarsitul primei replici. Proiectul nu s-a ocupat de gasirea interferentelor intre doua replici ce sunt legate de cuvinte cheie in mijlocul uneia dintre replici si nici cu gasirea relatiilor dintre propozitii ce fac parte din aceeasi replica. Cautarea e in mai multe faze: initial se cauta in replica principala (i.e. in cea cu id-ul mai mare deoarece pentru fiecare replica cautam legaturile puternice in alte replici anterioare) cuvinte cheie la inceputul frazei. Daca se gaseste unul consideram ca avem o legatura de interferenta intre replica principala si replica cu cea mai mare legatura cu aceasta (care se afla pe prima pozitie deoarece au fost ordonate anterior). In faza a doua se verifica toate replicile din lista ramase daca au cuvinte cheie la sfasitul lor. Daca da identificam relatia de interferenta dintre acestea si replica principala si le aduaugam.

Timpul de rulare pentru gasirea interferentelor date de replica principala este extrem de mic. Consumator de timp pentru modulul acesta este cautarea interferentelor

pentru replicile cu legaturi mai slabe (din acest motiv punem conditia suplimentara ca legatura sa fie penste un prag mai ridicat). Timpul de rulare in aceste conditii a fost dub 2 minute pentru 600 de replici cu aproximativ 300 de legaturi initiale in urma determinarii lanturilor lexicale.

Scrierea relatiilor de interferenta se face prin adaugarea unui tag <Interference> in interiorul tagului <Analysis>. Pentru fiecare legatura intre doua replici se adauga un nou tag <Infer> in interiorul tagului <Interference>. Descrierea interferentei se face in interiorul tagului <TextUT> ce contine:

- replica din care s-a extras tipul interferentei
- 3 atribute:
- id-ul primei replici
- id-ul celei de-a doua replici
- tipul interferentei

(exemplu: <TextUt from="103" to="157" type="conclusion">so in the training stage the method calculates w and b</TextUt>)

Topic Extraction

Pentru partea de extragere a topicurilor s-au aplicat 3 metode, descrise pe scurt mai jos. Toate metodele presupun parsarea textului din xml, linie cu linie. Fiecare linie este citita, apoi procesata. Liniile implicite introduse de programul de chat au fost eliminate (ex: ‚joins the room', ‚leaves the room',etc).

Metoda A. : Prima metoda, si cea mai cunoscuta este metoda frecventei. Se aplica in acest caz unigramelor (cuvintelor singulare). Mai jos voi descrie pas cu pas fiecare stagiu al algoritmului:

1. Citire text folosing xmlReader, linie cu linie
2. Pentru fiecare linie se elimina stopwords, apoi linia este impartita intr-o lista de cuvinte (split) folosind functia `stripStopWords_split()`. Aceasta functie returneaza lista cuvintelor in replica respectiva, excluzand stop-words.
3. Lista de cuvinte este apoi pasata ca parametru functiei `updateFreqList()`. Dictionarul `freq_dict` va juca rolul de tabela de frecventa, unde va incrementa fiecare noua aparitie a fiecarui cuvant. Cheia este cuvantul, valoarea o reprezinta frecventa.
4. Dupa terminarea parsarii liniilor si implicit a update-ului frecventelor, urmeaza sortarea dictionarului si taierea lui la primele *n* cuvinte dpdv. al frecventei: `sort_freq_dict(freq_dict,50)`. In acest caz am ales sa pastrez primele 50 de cuvinte, deoarece urmeaza 2 pasi de unificare.
5. Urmeaza procesarea efectiva prin apelul functiei `topicUnigrams(freq_dict)`. Aceasta functie incepe prin unificarea cuvintelor care au aceeasi radacina. Se apeleaza stemmer-ul pentru fiecare cuvant in parte care va fi verificat sa nu aiba acelasi stem cu restul cuvintelor dupa el (in acest mod se mai reduce din timpul de executie, desi complexitate secventei este $O(n^2)$). In caz ca se va intalni acelasi stem, al doilea cuvant va fi scos din dictionar, si frecventa primului cuvant va fi incrementata cu frecventa cuvantului scos.
6. Urmatorul pas este iterarea prin lista de sinonime si unificarea lor. Dupa ce se obtine lista de sinonime folosind functiile oferite de *WordNet* si anume `wn.synsets` si lemele oferite de lista de synset-uri, se trece la iterarea prin

lista de cuvinte, atasandu-se fiecaruia lista cu sinonime. Daca se gaseste un cuvant care are sinonim un alt cuvant sau un sinonim al acestuia, ultimul cuvant se sterge si frecventa primului i se incrementeaza proportional. Totodata, primului cuvant i se adauga si lista de sinonime preluata de la cel de-al doilea cuvant. Acest proces se repeta pana cand nu mai sunt cuvinte ce pot fi unificate.

7. Ultimul pas este transformarea dictionarului intr-o forma care poate fi usor scrisa in fisier – o lista de forma [frecventa_cuvant, cuvant, sinonim1, sinonim2, .. ,sinonimx]

Metoda B. : Se investigheaza frecventa bigramelor si trigramelor in text (desi sunt diferite, vom trata bigramele si trigramele la fel deoarece dpdv programatic nu exista diferente):

1. Se parseaza linie cu linie fisierul xml
2. Se elimina stopwords folosind functia `stripStopWords_nosplit()`, functie care nu returneaza o lista, ci un string care nu contine stopwords. Este necesar pentru a identifica bigramele/trigramele (de exemplu doua cuvinte separate prin punct nu pot forma o bigrama).
3. String-ul rezultat este pasat functiei `updateFreqList()`. Aceasta functie va actualiza tabela de frecvente pentru bigrame/trigrame.
4. Se sorteaza folosind functia `sort_freq_dict(freq_dict,15)`.
5. Se unifica bigramele, deoarece exista cazuri de bigrame care sunt in esenta identice, desi apar separat: ex: ‚svm method' cu ‚svn method**s**'. Tehnica este similara cu cea de la unigrame, folosind stemmer-ul.
6. In acest punct se creeaza tupluri de bigrame si frecventa lor, pentru a putea fi afisate si scrise in xml. Similar pentru trigrame. Functia folosita este: `getMultigramTopList(mgram,5)`, unde 5 este numarul de multigrame extrase in ordinea frecventei.

Metoda C. : Metoda experimentala, incearca sa investigheze cue-phrases. De exemplu va prelucra doar frazele care contin ‚about' sau incep cu ‚with'. De asemenea intrebarile mai pot fi investigate, sau propozitiile care contin cuvantul ‚method (of)'. De obicei astfel de propozitii sunt mult mai probabile sa introduca topicuri noi. Topicurile se pot identifica in urma analizei partilor de vorbire. Se cauta combinatii de substantive si adjective/adverbe, de marime 2 sau 3. Ex: NN NN sau RB NN, sau NN NN NN, etc. Aceasta metoda este doar experimentala deoarece depinde foarte mult de tipul de chat analizat. In acest moment nu am reusit decat un succes limitat. In schimb aceasta metoda a detectat prima de exemplu alaturarea ‚naive bayes'. Aceasta alaturare este greu de detectat, aproape imposibil prin metoda unigramelor (in wordnet nu exista conceptul ‚naive bayes' deci nu se poate unifica nici o pereche de cuvinte); doar prin bigrame s-a obtinut, si si aici doar datorita faptului de a fi repetata de 3 ori in text. Succestul unei astfel de metode variaza cu tipurile de texte analizate. Daca tipurile de texte sunt similare, rata de succes pentru o astfel de metoda poate fi foarte buna (cu conditia analizei umane a textului si a stabilitii frazelor introdctorii de topicuri noi)

Part Of Speech Tagging

Prin POST se realizeaza marcarea fiecarui cuvant din punct de vedere al partii de vorbire. Acest lucru este esential in cadrul proiectului, deoarece se pot determina relatii intre cuvinte, sau se pot alege doar anumite cuvinte ce sunt de un anumit tip ca parte de vorbire, etc.

Tagging-ul se poate realiza prin mai multe metode, prin expresii regulate, prin metode stocastice, metode hibride, etc. In cadrul proiectului am folosit un tagger hibrid, inlantuit. Dupa cum se poate observa in codul de mai jos, se foloste un tagger pe bigrame, daca se intampla sa nu fie recunoscut cuvantul, se foloste taggerul de Unigrame, apoi cel bazat pe expresii regulate, in final recurgandu-se la tagger-ul default. In acest mod se asigura o precizie cat mai mare a marcarii partii corecte de vorbire.

```
        self.default_tagger = nltk.DefaultTagger('NN')

        self.regexp_tagger = nltk.RegexpTagger(self.regpat, backoff =
self.default_tagger)

       self.unigram_tagger = nltk.UnigramTagger(brown_train, backoff =
self.regexp_tagger)

       self.bigram_tagger = nltk.BigramTagger(brown_train, backoff =
self.unigram_tagger)
```

Pentru a fi eficiente, tagger-ele trebuie antrenate. In acest caz am ales antrenarea pe corpusul brown de propozitii pre-marcate cu specific de stiri.

```
      brown_train = brown.tagged_sents(categories='news')[:500]
```

Gasirea Coreferintelor

Pentru gasirea coreferintelor s-a utilizat algoritmul de clusterizare Cardie – Wagstaff.

Particularitatile sale sunt legate de faptul ca fiind un algoritm ce face clusterizarea, poate rula pe seturi mari de date, nu are nevoie de antrenare si ruleaza in timp polinomial (dar nu prea eficient – este aproape $O(n^4)$). Un dezavantaj puternic este totusi faptul ca depinde foarte mult de raza aleasa pentru dimensiunea clusterelor, care teoretic ar trebui sa depinda de corpusul pe care se face clusterizarea.

In practica, Cardie – Wagstaff nu depinde atat de mult de aceasta distanta, ci mai mult de cat de bine se pot calcula distantele dintre doua entitati.

Algoritmul foloseste ca entitati NP – noun phrases, in care trebuie intai partitionate propozitiile ce se vor analizate. In principiu, se porneste de la o lista ordonata de astfel de NP obtinute din intrare: NP_1, NP_2, … NP_n. Aceasta lista se obtine pornind de la un tagging POST al string-ului de intrare, care e foarte important sa fie marcat corect. In plus, urmatorul pas este de asemenea critic: combinarea mai multor cuvinte intr-un singur NP.
Din pacate, acest pas al algoritmului este aproximat empiric prin forma cea mai generala pe care o pot avea conversatiile pe chat, deoarece nu exista o gramatica sau o expresie regulata care sa se poata folosi pe orice caz. Modul detaliat in care se realizeaza aceasta legare a mai multor cuvinte marcate intr-un singur NP poate fi vazut in functia *CorefRes.extractNP*.

Iesirea algoritmului este o lista de clustere (multimi) de NP care contin NP-urile ce sunt coreferente, conform criteriului de calcul al distantei dintre doua NP-uri:

$$dist(NP_i, NP_j) = \sum_{f \in F} w_f * incompatibility_f(NP_i, NP_j)$$

Aceasta distanta se calculeaza ca o suma dupa o lista de features ale NP-urilor, fiecare feature avand un weight (pondere) si o functie asociata prin care se poate masura incompatibilitatea intre doua NP pe baza acestui feature.

Functia de incompatibilitate intoarce o valoare in intervalul [0;1].

Lista de features este alcatuita din 12 atribute:

1. Cuvintele folosite (words): are ponderea de 10.0, iar functia se calculeaza ca raportul numar cuvinte ce nu se potrivesc / numarul cuvinte in NP-ul de lungime maxima

2. Cuvantul din capat (head noun): deoarece un NP poate fi alcatuit din mai multe cuvinte: substantive (N – noun), pronume (PP – pronouns), articole (AT – articol nedefinit sau DT – definit), se ia ultimul cuvant. Ponderea este de 1.0 iar functia de incompatibilitate intoarce 1 daca difera si 0 in rest.

3. Pozitia in string: fiecare NP are o pozitie asociata in lista de intrare, iar ponderea acestui atribut este 5.0 . Functia de incompatibilitate este raportul dintre valoarea absoluta a diferentei de pozitie intre NPi si NPj si distanta maxima posibila (care e numarul de NP din lista de intrare).

4. Tipul de pronume: NOMinative, ACCusative, POSSessive, AMBIguous sau NONE. Ponderea este egala cu raza aleasa pentru algoritm. Functia de incompatibilitate este 1 daca NPi e pronume dar NPj nu e si 0 in rest.

5. Tipul de articol: DEFinite, INDEFinite, NONE. Ponderea = r. f = 1 daca NPj este nedefinit si apozitional, 0 in rest. Un NP este apozitional daca:
    - are un articol
    - se afla intre virgule in textul initial
    - dupa el urmeaza imediat un alt NP

6. Subsumare de cuvinte: pondere –infinit, f = 1 daca NPi subsumeaza NPj, adica orice cuvant din NPj se gaseste in NPi.

7. Apozitional: pondere –infinit, f = 1 daca NPj este apozitional iar NPi este predecesorul lui

8. Numarul: 1 pentru singular, 2 pentru plural, pondere = +infinit, f = 1 daca nu au acelasi numar

9. Nume Propriu: 1 daca da, 0 daca nu, pondere = +infinit, f = 1 doar daca ambele sunt nume proprii si niciunul din cuvinte nu se potriveste intre ele

10. Clasa semantica: ‘time’, ‘city’, ‘animal’, ‘human’, ‘object’, ponderea = +infinit, f = 1 daca nu au aceeasi clasa

11. Genul: ‘M’, ‘F’, ‘E’, ‘N’ : masc, feminin, oricare (either), neutru. ponderea = +infinit, f = 1 daca nu au acelasi gen (dar either poate face match pe orice).

12. Animat: 0 pentru lucruri lipsite de viata sau 1, daca e din clasa ‘animal’ sau ‘human’. Pondere si functie similare celor anterioare.

Pentru stocarea unui NP si a informatiilor legate de el (atribute, context, etc) se foloseste clasa NP, iar o instanta se initializeaza cu

```
def __init__(self, tagtext, context, pos)
```

unde tagtext este o lista de tupluri tagged: (‘cuvant’, ‘tag gramatica’) care intra in alcatuirea unui singur NP. Contextul este un dictionar cu doua chei: {tagged: , text: }. Cheia text este stringul initial. Cheia tagged contine lista obtinuta prin extractNP din clasa CorefRes, lista ce contine impartirea in NP, si tagtext, implicit.

Urmatorul pas dupa instantierea unei NP este modificarea razei implicite de 2.0:

```
def setRadius (self, rad):
```

Apoi, putem trece la calcularea tuturor valorilor atributelor, care se face prin

```
def updateValues (self):
```

Fiecare atribut are functia proprie definita pentru calculul incompatibilitatii, numite de la f1 la f12 si apelabile cu un argument alt NP.

In final, se poate calcula distanta dintre 2 NP-uri dupa modelul:

a = NP(…)
b = NP(…)
a.setRadius(50.0)
b.setRadius(50.0)
a.updateValues()
b.updateValues()
print a.dist(b)

Din acest moment, se pot folosi NP-uri in algoritmul implementat in clasa CorefRes, in metoda

```
def clusterize (self, string, radius):
```

ce primeste ca parametru un string cu mai multe propozitii si raza dorita la clusterizare si intoarce ca rezultat o lista cu 2 elemente:

- lista de NP rezultata
- o lista de indici cu clusterul corespunzator fiecarui NP din lista

Indicii de cluster nu sunt normalizati, in sensul ca lista va avea un continut similar cu [2,7,4,2,4,4], existand deci trei clustere cu indicii 2,4,7. In outputul pentru XML, aceste numere vor fi normalizate.

Cardie_Wagstaff (NP_1, NP_2, NP_3… NP_n):
1. Fie r raza de clusterizare
2. Clusterul initial $c_i = \{NP_i\}$ pentru fiecare i=1,n
3. Pentru fiecare NP_j incepand cu n, n-1, … 1, si toti precedentii sai NP_i:
    4. Fie $d = dist(NP_i, NP_j)$
    5. Fie c_i = cluster NP_i, c_j = cluster NP_j
    6. Daca $d < r$ si Toate_NP_Compatibile(c_i, c_j) atunci:
        7. $c_j = c_i + c_j$ (reuniune clustere)

Toate_NP_Compatibile (c_i, c_j):
1. Pentru toate NP_a din c_j:

2. Pentru toate NP_b din c_i:
    3. Daca dist(NP_a, NP_b) = - infinit atunci:
        4.return False
5. return True.

Din algoritm se observa ca avem o bucla exterioara in pasul 3 cu j de la n…1, inca una dupa i de la 1 la j (n*(n-1)/2 pasi) iar in interior se apeleaza functia Toate_NP_Compatibile care contine iar complexitate $O(n^2)$, deci in total $O(n^4)$, unde n = numarul initial de NP.

Complexitatea cuadratica nu este foarte potrivita pentru un chat de 600 de replici, fiecare cu aproximativ 5 NP-uri, iar daca adaugam si complexitatea accesarii WordNet si a calculului din spatele functiilor din clasa NP, se ajunge usor la timpi crescuti de analiza.

Din cauza acestui aspect, analiza coreferintelor se face pe perechi de 2 replici concatenate, folosind informatia de refid din replici, deja existenta de pe parcursul sesiunii de chat. Practic, utilizatorii marcheaza o replica ce are o referinta la una anterioara, iar in timpul analizei XML-ului de input, se gaseste aceasta replica anterioara, se concateneaza cu cea curenta, si pe stringul rezultat se aplica CorefRes.corefXML() care va introduce in output urmatoarea structura DOM:

```
<Analysis>
<Coref>
      <CorefUt>
            <RefUt1>14</RefUt1>
            <RefUt2>8</RefUt2><NP id="1">hi/NN </NP><NP id="2">i/NN
</NP><NP id="3">m/NN </NP><NP id="4">Stefan/NN </NP><NP
id="5">Dumitrescu/NN </NP><NP id="6">nick/NN </NP><NP id="7">stefan/NN
</NP><NP id="8">i/NN </NP><NP id="9">the/AT honor/NN </NP><NP
id="10">defend/NN </NP><NP id="11">naive/NN </NP><NP id="12">bayes/NNS
</NP>
            <Cluster id="1"><NPid id="1"/><NPid id="2"/><NPid
id="3"/><NPid id="6"/><NPid id="8"/><NPid id="9"/><NPid
id="10"/></Cluster>
            <Cluster id="2"><NPid id="4"/><NPid id="5"/><NPid
id="7"/><NPid id="11"/></Cluster>
            <Cluster id="3"><NPid id="12"/></Cluster></CorefUt>
```

<Analysis> apare ca element in <Body>, si va contine un element radacina pentru aceasta parte de analiza, numit <Coref>. In el, pentru fiecare pereche analiza de replici, se face un element <CorefUt>, ce contine:

- idurile replicilor analizate, in <RefUt1> <RefUt2>
- lista de NP-uri: <NP>, fiecare cu atributul id ce arata pozitia lor, si textul tagged
- lista de clustere rezultata, fiecare cluster cu id-ul normalizat, si o lista de elemente <NPid> pentru fiecare NP ce se afla in cluster. Atributul id din <NPid> pointeaza la id-ul unui element <NP> precizat anterior.

Ar trebui precizat in final si modul in care se calculeaza unele din valorile pentru atribute, deoarece precizia algoritmului depinde foarte mult de asta:

- numarul se determina verificand daca exista un ‘s’ la finalul cuvantului

- clasa semantica se determina folosind functia wordnet.wup_similarity(synset1, synset2) si gasind maximul valorilor. Din nefericire, nu functioneaza perfect, incadrand uneori nume proprii in clasa animalelor, in loc de human. Pentru

verbe, trebuie folosit path_similarity, deoarece wup nu functioneaza decat pe substantive.

- genul nu se poate determina in prezent, deoarece nu exista functii in NLTK, iar orice incercare de a gasi similaritatea intre un cuvant si conceptul de ‘masculine’ si ‘feminine’ returneaza aceeasi valoare, indiferent de functia folosita.

- apozitionalitatea nu se verifica deoarece se poate demonstra experimental ca frecventa de aparitie in sesiuni de chat este sub 5%. Se presupune din start ca apozitionalitatea e = 0.

Rezultatele algoritmului sunt puternic afectate de aceste restrictii asupra modului automat de calcul al acestor valori de atribute. De notat faptul ca pentru functionarea corecta, are nevoie de text POST manual, cu toate adnotarile pentru gen, clasa semantica, etc.

Dezvoltari ulterioare

- Pentru partea de POST se poate imbunatati tagger-ul daca se creeaza niste expresii regulate mai potrivite sesiunilor de chat, pornind de la exemple.
- Coreferintele pot fi analizate prin mai multi algoritmi pentru eliminarea false-positives
- Se poate folosi un corpus de text adnotat pentru algoritmul de clusterizare, lucru ce ar spori precizia
- Pentru partea de interferente in viitor trebuie dezvoltat un algoritm pentru gasirea interferentelor dintre replici cand cuvintele cheie se afla in interiorul replicilor.

Concluzii

Dupa observarea modului de rulare al majoritatii algoritmilor, am observat ca in majoritatea cazurilor s-a obtinut rezultatul asteptat prin metodele automate. Exista cateva exceptii: POST si Coreferintele nu functioneaza suficient de precis, ceea ce era de asteptat.

Training-ul la POST este foarte important pentru buna evolutie a algoritmului, care depinde de un set mare de propozitii deja tagged din corpusuri existente. Deoarece noi am dorit pastrarea limitelor normale de timp la antrenarea tagger-ului, am folosit doar 500 de replici, ceea ce nu e suficient, dupa cum s-a observat pentru exemple de verbe care sunt marcate ca si substantive comune la plural NNS. Precizia estimata este 89.6%.

Algoritmul de Coreferinte nu este adaptat suficient lucrului pe text provenit din chat intre 4 persoane, deoarece o replica poate referi o alta la distanta de 100 de replici intermediare, lucru ce duce la o crestere exponentiala a timpului petrecut in algoritm.

In plus, algoritmul depinde si de un POST complet corect, cu genurile specificate pentru fiecare substantiv/pronume, si cu feature-urile gasite perfect: in ceea ce priveste clasa

semantica, este greu de evaluat in cod daca un cuvant este din clasa timp, animal, om, etc, din cauza ca functiile ce calculeaza similitudinea dau gres pe exemple banale.

Proiectul nostru arata tehnici care se pot imbunatati in viitor.

Bibliografie

1. Virtual Math Teams Project
http://www.ipsi.fraunhofer.de/concert/index_en.shtml?projects/vmt

2. NLTK Book, *Steven Bird, Ewan Klein, Edward Loper*
http://www.nltk.org/book

3. NLTK API Docs
http://nltk.googlecode.com/svn/trunk/doc/api/index.html

4. Python Docs
http://docs.python.org/

5. Clustering Algorithms for Noun Phrase Coreference Resolution, *Roxana Angheluta, Patrick Jeuniaux, Rudradeb Mitra, Marie-Francine Moens*

6. Noun Phrase Coreference as Clustering, *Claire Cardie, Kiri Wagstaff*

7. CorefDraw: A Tool for Annotation and Visualization of Coreference Data, *Stefan Trausan-Matu, Sanda Harabagiu, Razvan Bunescu*

8. Coreference Resolution: A Survey, *Pradheep Elango*

9. Using Cue Phrases to Determine a Set of Rhetorical Relations - *Alistair Knott*

10. Lexical Chains and Semantic Distance - Eurolan-2001, August 2001, Iasi, Romania