**Coperta**

Cuprins

[I. Prezentare temă 4](#_Toc41524491)

[II. Considerații teoretice – algoritmi de învățare 4](#_Toc41524492)

[III. Consideraţii practice 7](#_Toc41524493)

[3.1 Descriere generală proiect 7](#_Toc41524494)

[3.2 Descriere separată bloc 8](#_Toc41524495)

[3.2.1 Dataset 8](#_Toc41524496)

[3.2.1.1 Impartire 70% - setul de antrenament, 30% - setul de testare 10](#_Toc41524497)

[3.2.1.2 Cross-Validation 10](#_Toc41524498)

[3.2.2 Preprocessing 14](#_Toc41524499)

[3.2.2.1 Sentence tokenization 14](#_Toc41524500)

[3.2.2.2 Part of speech classifier 16](#_Toc41524501)

[3.2.2.3 Data cleaning & normalization 20](#_Toc41524502)

[3.2.3 Model 22](#_Toc41524503)

[3.2.3.1 Hidden Markov Model 22](#_Toc41524504)

[a) Emission probability 22](#_Toc41524505)

[b) Transition probability 26](#_Toc41524506)

[3.2.2.1 Unknown words model 34](#_Toc41524507)

[3.2.4 Decoder 46](#_Toc41524508)

[3.2.4.1 Forward method 48](#_Toc41524509)

[3.2.4.2 Backward method 56](#_Toc41524510)

[3.2.4.3 Bidirectional method 58](#_Toc41524511)

[3.2.5 Evaluation 59](#_Toc41524512)

[3.2.5.1 Naive accuracy 59](#_Toc41524513)

[3.2.5.2 Confusion matrix 60](#_Toc41524514)

[IV. Rezultate 65](#_Toc41524515)

[V. Anexa 1: Exemplu etichetare parte de vorbire 65](#_Toc41524516)

[VI. Bibliografie 65](#_Toc41524517)

# I. Prezentare temă

# II. Considerații teoretice – algoritmi de învățare

Ecuatia cu decoder produs tranzitie \* emisie

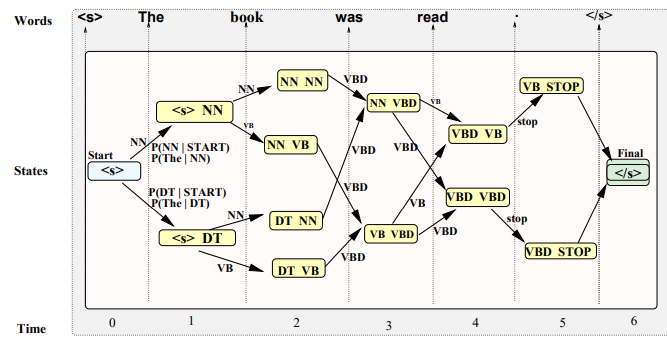
****

Figura x - Reprezentare stări trellis [5]

****

Figura x - Pseudocod Viterbi general [3]

****

Figura x – Pseudocod Viterbi pentru etichetarea părţii de vorbire [1]

# III. Consideraţii practice

## 3.1 Descriere generală proiect

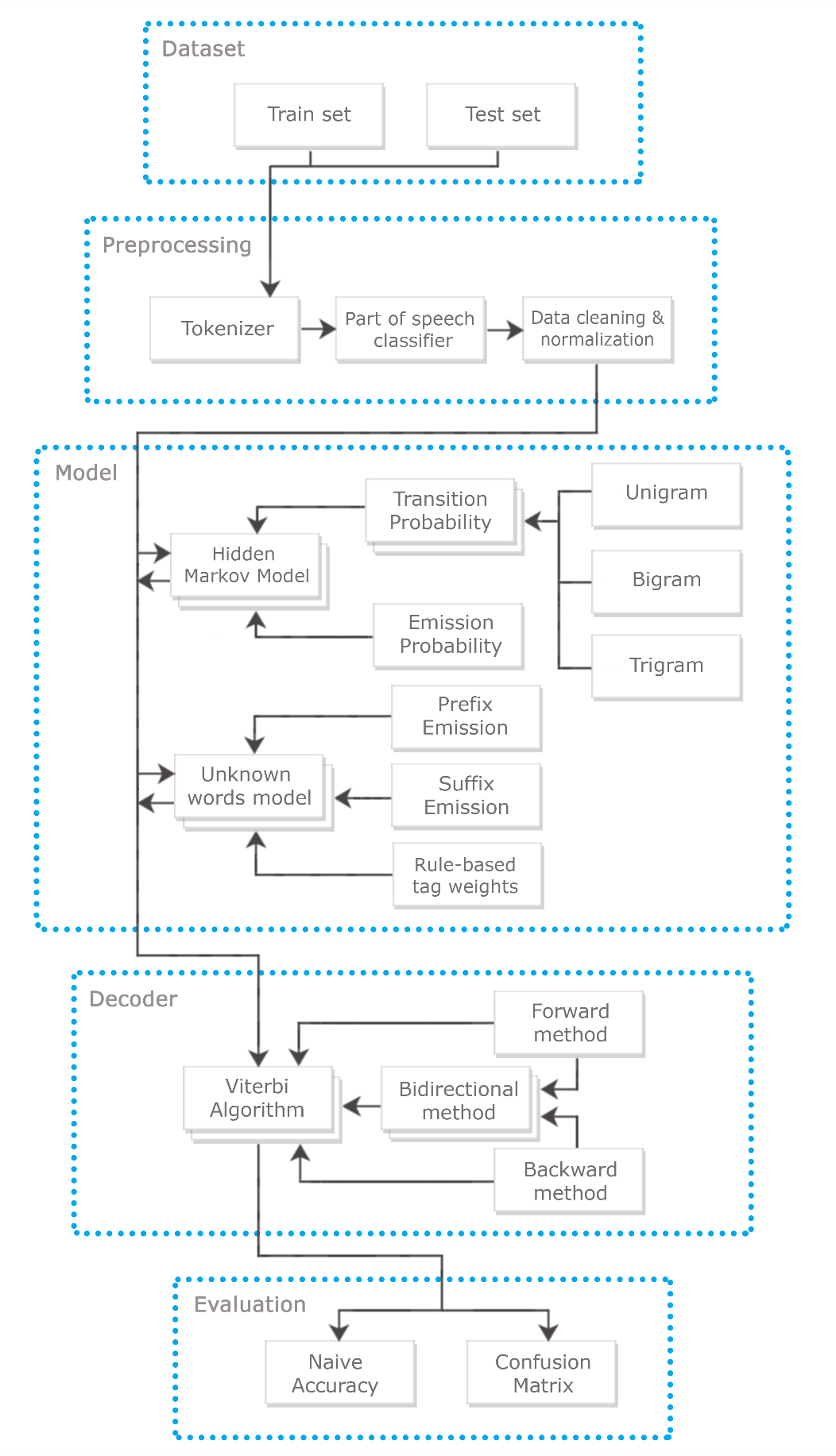


Figura 3.1 - Arhitectura sistemului de etichetare

Se poate observa că sistemul este structurat pe 5 mari categorii: Dataset, Preprocessing, Model, Decoder și Evaluation. În prima componentă, Dataset, este descris setul de date pe care se antrenează  sistemul și pe care este testat pentru a vedea rezultatele finale ale algoritmului. Acesta descrie și 2 metode de antrenare diferite, una bazată pe antrenare și testare pe aceleași fișiere, cealaltă bazată pe cross-validation. În Preprocessing se descriu metode de curățare, integrare, transformare, reducere și discretizare a setului de date astfel încât la final, datele procesate să nu ofere informații eronate sau rezultate false/greșite. Această componentă include și tokenizarea și clasificarea în taguri generale a setului de date.

În Model, sunt descriși algoritmii de învățare care modelează arhitectura sistemului, acesta este format din 2 modele majore precum Hidden Markov Model (HMM) și Unknown Words Model (model pentru cuvintele necunoscute). Modelul este cea mai importantă componentă din structura, fără aceasta nu s-ar putea predicționa tagul cuvintelor. În Decoder, este descris algoritmul dinamic recursiv a lui Viterbi și metodele implementate cu acesta. Viterbi este un algoritm pentru decodificarea secvențelor stărilor ascunse într-un model de tip HMM.

În ultima componentă, Evaluation, este evaluat algoritmul de clasificare/predicție, în funcție de diferitele metrici de evaluare precum acuratețea, precizia, recall-ul, f-measure-ul și specificitatea. În secțiunile următoare, este descrisă detaliat fiecare componentă și subcomponentă a ei + codul implementat (în limbaj de programare), specific fiecărei secțiuni iar în anexa 1 este prezentat un exemplu complet de utilizare a formulelor prezentate la descrierea blocurilor separate.

Proiectul a fost scris în limbajul de programare C# .NET, target framework: .NET Core 2.1. Pe pagina de github a proiectului [6], se află toată istoria modificărilor, sursele codului, modele salvate în format JSON, unit-testuri pentru fiecare clasă, dataset-ul folosit, diferite statistici, evaluări și documentația finală.

## 3.2 Descriere separată bloc

### 3.2.1 Dataset

Setul de date (numit si corpus) este o colectie de date prelucrate si alese special pentru a putea evalua calitatea sistemului de etichetare. Pentru setul de date se va folosi **Brown Corpus**, o colectie de propozitii si fraze in limba engleza colectate si organizate de W. Nelson Francis & Henry Kucera din departamentul lingvistic de la Universitatea Brown. Colectia aceasta are peste 1 milion de cuvinte in total si contine exact 500 de documente [7]. Cele 500 de documente sunt impartite in 2 mari categorii, prima categorie fiind proza informativa cu urmatoarele sub-categorii:

A. Presă: Reportaje – 44 documente

B. Presă: Editorial – 27 documente

C. Presă: Recenzii (teatru, carti, muzica, dans) – 17 documente

D. Religie – 17 documente

E. Skill-uri si hobby-uri – 36 documente

F. Folclor popular – 48 documente

G. Scrisori, bibliografii, biografii – 75 documente

H. Diverse – 30 documente

J. Articole stiintifice– 80 documente

TOTAL – 374 documente

iar a doua categorie fiind proza imaginativa cu urmatoarele sub-categorii:

K. Fictiune generala – 29 documente

L. Mister și ficțiune detectiva – 24 documente

M. Opere științifico-fantastice – 6 documente

N. Aventura și ficțiunea western – 29 documente

P. Povesti romantice și de dragoste – 29 documente

R. Umor – 9 documente

TOTAL – 126 documente

Fiecare document are peste 2000 de cuvinte iar fiecare cuvant si caracter este delimitat de un slash ‘/’, urmat de tagul aferent partii de vorbire al acestuia, sub forma “cuvant/tag”. Setul de taguri folosit este numit **Penn Treebank**, iar acesta contine 45 de taguri:



Figura 3.2 – Penn Treebank tagset (Jurafsky slp3-post, 2019)

Acest set de date este folositor pentru un algoritm de *invatare supervizata pe o clasa* sau *one class classification*. Exemplu de propozitie in setul de date:

ex. *Mr./np Remarque's/np$ conception/nn of/in this/dt novel/nn was/bedz sound/jj and/cc perhaps/rb even/rb noble/jj ./.*

Exista peste 100 de taguri (~103) individuale in Brown Corpus, multe fiind derivate de la forma de baza a partii de vorbire, de exemplu tagul *np$* este substantiv propriu-zis posesiv la singular, fiind derivat din substantiv. Unele taguri pot aparea combinate cu delimitatorul ‘+’:

ex. … Y'all/ppss ***wanna/vb+to*** walk/vb …

, deoarece *want/vb* + *to/to 🡺 wanna/vb+to.*

Alte taguri pot aparea formate din tagul propriu-zis si un tag prefix de indicatie a unei informatii suplimentare folosind delimitatorul ‘-‘, ca fie este un cuvant preluat din alta limba ‘FW’ (foreign word), fie ca acel cuvant apare in titlu ‘TL’ sau alte informatii de forma aceasta care ar putea fi importante la clasificare.

ex. … yesterday/nr ***en/fw-in route/fw-nn*** to/in his/pp$ …

*, unde /fw-in* este tagul pentru prepozitie intr-o limba straina si */fw-nn* estesubstantiv la singular intr-o limba straina.

In unele tag-uri pot aparea si simboluri precum ‘$’(cuvant la posesiv), ‘\*’(cuvinte negate precum can’t, wouldn’t, shouldn’t), ‘nil’(nespecificat)

**Train & test set**

Inainte ca modelul de predictie sa fie folosit pe date reale acesta mai intai este evaluat. Pentru a realiza evaluarea modelului, trebuie sa impartim setul de date intr-un set de antrenament (train set) si un set de testare (test set). Exista 2 metode de impartire a setului de date din proiect:

#### 3.2.1.1 Impartire 70% - setul de antrenament, 30% - setul de testare

Aceasta metoda este oarecum evidenta, pentru fiecare sub-categorie din Brown Corpus, se aleg din documentele aferente acesteia, 70% documente pentru antrenare si 30% documente pentru testare.

ex. Pentru sub-categoria *J. Articole stiintifice,* avem in total 80 de documente, primele 56 documente (70%) le vom folosi pentru antrenare iar ultimele 24 documente (30%) le vom folosi pentru testare.

#### 3.2.1.2 Cross-Validation

Numita si “rotation estimation”, este o tehnica de validare a modelului, pentru a vedea rezultatul generalizat al modelului pentru un set de date independent. Acest mod de validare este folositor pentru a vedea abilitatea modelului la predictia pentru date noi, inlaturand probleme precum “overfitting” sau “selection-bias”, probleme care presupun impartirea setului de date astfel incat pentru o portiune aleasa din setul de antrenament ar da predictii foarte mici daca ar fi considerata set de testare sau de validare.

**K folds cross-validation**

Acesta presupune mai intai alegerea unui numar K (deobicei K=4 sau K=10) si apoi impartirea setului de date in K folds. Dupa aceasta impartire, se va itera setul de date astfel incat fiecare fold individual sa fie set de testare si restul set de antrenare. Dupa fiecare evaluare a setului de testare se va salva acuratetea modelului pt. acel fold si se va pastra pana cand vom evalua toate fold-urile.



Figura 3.3 – exemplu cross-validation k = 4 (Quantinsti Cross-validation, 2019)

La sfarsit dupa ce am evaluat fiecare fold in parte, se va calcula media aritmetica pentru fiecare fold pentru a obtine o acuratete generala pe model.

|  |  |
| --- | --- |
|  | (3.1) |
|  |  |

k - numarul de fold-uri alese

– acuratetea pe fold-ul i

**Implementare K folds cross-validation**

Functia principala pentru cross-validation are 3 parametrii de intrare: *filePath* pentru path-ul la folder-ul cu documentele Brown Corpus, *fold* pentru numarul de fold-uri folosite pentru impartire a fisierelor (parametru default = 4) si shuffle pentru optiunea booleana daca a amesteca documentele sau a le lasa cum apar in director (parametru default = False)

**private** void SetFilesForCrossValidation**(**string filePath**,**

int fold **=** 10**,**

bool shuffle **=** **false)**

**{**

List**<**string**>** files **=** FileReader**.**GetAllTextFromDirectoryAsList**(**filePath**);**

int filesPerFold **=** files**.**Count **/** fold**;**

**this.**TestFile **=** **new** string**[**fold**];**

**this.**TrainFile **=** **new** string**[**fold**];**

**if** **(**shuffle**)**

files **=** **this.**Shuffle**(**files**);**

**for** **(**int crossIndex **=** 0**;** crossIndex **<** fold**;** crossIndex**++)**

**{**

var IndividualTrainFiles **=** **new** List**<**string**>();**

var IndividualTestFiles **=** **new** List**<**string**>();**

**for** **(**int i **=** 0**;** i **<** files**.**Count**;** i**++)**

**{**

**if** **(**i **>=** **(**filesPerFold **\*** crossIndex**)** **&&**

i **<** **(**filesPerFold **\*** **(**crossIndex **+** 1**)))**

**{**

IndividualTestFiles**.**Add**(**files**[**i**]);**

**}**

**else**

**{**

IndividualTrainFiles**.**Add**(**files**[**i**]);**

**}**

**}**

string trainf **=** String**.**Join**(**" "**,** IndividualTrainFiles**);**

string testf **=** String**.**Join**(**" "**,** IndividualTestFiles**);**

**this.**TrainFile**[**crossIndex**]** **=** trainf**;**

**this.**TestFile**[**crossIndex**]** **=** testf**;**

**}**

**}**

In functia aceasta, se vor citi toate documentele din folderul cu fisierele Brown si se vor returna textul acestora ca lista pentru fiecare document separat (files.count = 500).

In continuare se vor initia vectorii proprietati ai clasei pentru fisierele de antrenare si de testare, se va calcula cate fisiere sunt per fold (de exemplu pentru fold = 4, vor exista 125 documente per fold) si dupa aceea se va verifica daca optiunea de shuffle a fost “activata”, daca da atunci lista de fisiere cu documente Brown va fi amestecata.

In for-ul din functia principala, se va itera de la 0 pana la fold number ales, se va calcula lista individuala de fisiere pt. antrenare si testare si dupa care din aceste liste se va concatena tot continutul acestora prin simbolul char de spatiu si rezultatul se va salva la crossIndex-ul fold-ului in vectorul propietate de antrenare si de testare.

Pentru a intelege mai bine conditia evidentiata, luam ca exemplu filesPerFold = 125, crossIndex = 3 (adica ultimul fold), atunci aceasta conditie s-ar traduce => i(index-ul pentru document) mai mare sau egal ca 375 si i mai mic strict ca 500, ceea ce reprezinta 25% pentru setul de testare salvat in lista *IndividualTestFiles* iar celelalte fisiere de la 0 la 374 adica restul de 75% se vor salva in lista de antrenare *IndividualTrainFiles*.

Pentru obtinerea textului din foldere in lista avem urmatoarea implementare:

List**<**string**>** outputFile **=** **new** List**<**string**>();**

var files **=** Directory**.**GetFiles**(**inputDir**);**

**foreach** **(**string file **in** files**)**

**{**

string elem **=** GetTextFromFileAsString**(**file**);**

outputFile**.**Add**(**elem**);**

**}**

GetTextFromFileAsString este echivalent cu metoda din File, ReadAllText(string path).

Functia de amestecare implementeaza algoritmul *Fisher–Yates Shuffling,* acesta alege un numar random intre index-ul de la primu element din lista si ultimul element din lista si incepe sa numere pana la acesta, ignorand elementele din lista deja folosite in amestecare (scratched elements), odata ajuns la index-ul respectiv, va prelua elementul de acolo si il va adauga la finalul listei nou formate, punand astfel un flag de folosit pt. indexul respectiv si decrementand numarul total de numere.

ex.Fisher–Yates: lista cu numere: [1, 2, 3, 4, 5], n = 5

alegem un numar random de la 1 la n (1 fiind index-ul la primul element din lista, n fiind index-ul la ultimul element din lista)

Random = 2 🡺 noua\_lista = [2], lista\_veche = [1, 2, 3, 4, 5], n =4

Random = 3 🡺 noua\_lista = [2, 4], lista\_veche = [1, 2, 3, ~~4~~, 5], n =3

Random = 1 🡺 noua\_lista = [2, 4, 1], lista\_veche = [1, 2, 3, 4, 5], n =2

Random = 2 🡺 noua\_lista = [2, 4, 1, 5], lista\_veche = [1, 2, 3, 4, 5], n =1

noua\_lista\_finala = [2, 4, 1, 5, 3]

Implementarea acestuia este destul de ineficienta si complicata, pentru algoritm ar trebuii sa folosim 2 liste si sa avem si un flag pentru indexul elementelor pe care le-am folosit deja, dar exista totusi o metoda mai usoara de implementare, anume *Durstenfeld's version.* Acesta presupune interschimbarea elementului la index-ul n cu elementul la index-ul ales random si decrementarea lui n dupa fiecare interschimbare.

ex. Durstenfeld’s version: lista: [1, 2, 3, 4, 5], n = 5

Random = 2 🡺 dfeld\_list = [1, 5, 3, 4 | 2], n = 4

Random = 3 🡺 dfeld\_list = [1, 5, 4 | 3, 2], n = 3

Random = 3 🡺 dfeld\_list = [1, 5 | 4, 3, 2], n = 2

Random = 1 🡺 dfeld\_list = [5 | 1, 4, 3, 2], n = 1

dfeld\_list\_final = [5, 1, 4, 3, 2]

Algorimtul in versiunea lui Durstenfeld(Fisher-Yates Shuffling solutions, 2009)):

**Private**List**<**string**>** Shuffle**(**List**<**string**>** list**)**

**{**

Random rng **=** **new** Random**();**

int n **=** list**.**Count**;**

**while** **(**n **>** 1**)**

**{**

n**--;**

int k **=** rng**.**Next**(**n **+** 1**);**

string value **=** list**[**k**];**

list**[**k**]** **=** list**[**n**];**

list**[**n**]** **=** value**;**

**}**

**return** list**;**

**}**

### 3.2.2 Preprocessing

Preprocessing este procesul in care curatam si normalizam datele astfel incat sa ne fie mai usor de lucrat cu acestea. Lipsa acestui proces poate ingreuna procesul de invatare si poate influenta calitatea rezultatelor. Pentru acest sistem, procesul de preprocessing include si despartirea cuvintelor in lista si clasificarea intr-un tag general. Cele 3 procese aici sunt:

* Sentence tokenization
* Part of speech classifier
* Data cleaning & normalization

#### 3.2.2.1 Sentence tokenization

Tokenization este procesul de a delimita cuvintele dintr-un text si posibil a le clasifica, folosit in analiza lexicala. Acesta deobicei delimiteaza pe baza unei reguli, algoritmul folosit pentru model este *Whitespace Tokenizer* dar se pot implementa mai multi algoritmi de tokenizare pentru model, pentru cazul in care modelul va fi aplicat pe date si texte din lumea reala.

Algoritmul *Whitespace Tokenizer* imparte un string intr-o singura lista de string-uri, si fiecare string este delimitat atunci cand gaseste caracterul de spatiu, tab sau newline (coduri ASCII: 32, 09, 13).

ex. I’m home but he is not home.

Token\_list = [“I’m”, “home”, “but”, “he”, “is”, “not”, “home.”]

**Implementare sentence tokenization**

Algoritmul whitespace tokenizer este echivalent cu functia Split(‘ ‘) din c#:

**public** static List**<**string**>** TokenizePennTreebank**(**string Text**)**

**{**

List**<**string**>** tokenizedText **=** **new** List**<**string**>();**

string word **=** ""**;**

**foreach(**char c **in** Text**)**

**{**

**if** **(!**Char**.**IsWhiteSpace**(**c**))**

word **+=** c**;**

**else** **if** **(!**String**.**IsNullOrEmpty**(**word**))**

**{**

tokenizedText**.**Add**(**word**);**

word **=** ""**;**

**}**

**}**

**if** **(!**String**.**IsNullOrEmpty**(**word**))**

tokenizedText**.**Add**(**word**);**

**return** tokenizedText**;**

**}**

Acest tokenizer este cel mai potrivit deoarece, in setul de date, fiecare cuvant+tag este delimitat de spatiu, deci lista de token-uri va arata de forma: list = […., “he/pn”, “made/vbd”, “pancakes/nn”, “./.” , ….].

Totusi se prefera despartirea cuvantului de tag, pentru asta se implementeaza urmatoarea structura:

**public** struct WordTag

**{**

**public** string word**;**

**public** string tag**;**

**public** WordTag**(**string word**,** string tag**)**

**{**

**this.**word **=** word**;**

**this.**tag **=** tag**;**

**}**

**}**

, iar codul de despartire si adaugare in lista:

**public** static List**<**WordTag**>** SeparateTagFromWord**(**List**<**string**>** Words**)**

**{**

List**<**WordTag**>** wordTags **=** **new** List**<**WordTag**>();**

**foreach** **(**var word **in** Words**)**

**{**

string**[]** separated **=** word**.**Split**(**'/'**);**

string separatedTag **=** separated**[**separated**.**Length **-** 1**];**

string combWord **=** ""**;**

**if** **(**separated**.**Length **>** 2**)**

**{**

**for** **(**int i **=** 0**;** i **<** separated**.**Length **-** 1**;** i**++)**

combWord **+=** separated**[**i**]** **+** "/"**;**

combWord **=** combWord**.**Remove**(**combWord**.**Length **-** 1**);**

**}**

**else** combWord **=** separated**[**0**];**

wordTags**.**Add**(new** WordTag**(**combWord**,** separatedTag**));**

**}**

**return** wordTags**;**

**}**

Separam cuvantul de tag prin functia split cu parametrul ‘/’ si obtinem un vector de string-uri. Ultimul string din vector va fi mereu tag-ul cuvantului dar pentru restul vectorului nu exista aceeasi siguranta, de exemplu cuvintele compuse precum “input/output” pot aparea in setul de date si daca acestea nu sunt salvate integral, pot aduce rezultate gresite in procesul de invatare. Se cunoaste ca in cazul cel mai bun vom avea un vector de string-uri de dimensiune = 2 (cuvant necompus+tag) iar in cazul opus, in conditia evidentiata de mai sus, se itereaza vectorul de cuvinte si se concateneaza tot ce apare in vector pana la tag, dupa care se sterge ultimul caracter(care va fi ‘/’) din string-ul *combWord* obtinut, deoarece dupa fiecare cuvant se adauga caracterul ‘/’. La sfarsit se va adauga in lista de structuri, string-ul *combWord* care contine cuvantul compus/necompus si string-ul *separatedTag* care contine tag-ul cuvantului.

#### 3.2.2.2 Part of speech classifier

Pentru o mai buna predictie, se alege clasificarea partilor de vorbire in x categorii, acestea fiind partile de vorbire de baza pentru a acoperi majoritatea tag-urilor din Brown Corpus. Acest proces nu este unul automat facut de un algoritm, ci este facut in urma unei analize a partilor de vorbire din engleza si alese cele mai potrivite pentru setul de date cu care se lucreaza (Brown Corpus). Pentru x = 10, am ales urmatoarele categorii:

1. Noun (NN) – substantive
2. Pronoun (PN) - pronume
3. Verb (VB) – verb
4. Adjective (JJ) – adjectiv
5. Adverb (RB) – adverb
6. Preposition (PP) – prepozitie
7. Conjuction (CC) – conjunctie
8. Article/Determiner (AT/DT) – articol & determinant (cel din engleza)
9. End of sentence (.) – sfarsit de propozitie, indiferent de semne ‘!’, ‘?’, “.” etc.
10. Others (OT) – alte parti de vorbire cum ar fi interjectie, cardinal numeral precum “six”, “two”, cuvinte din alta limba etc.

Deci in continuare, orice cuvant/string din lista de antrenare sau de testare va avea doar un singur tag din cele 10 mentionate anterior.

In urma clasificarii partiilor de vorbire, obtinem urmatoarele rezultate statistice:



Figura 3.4 – Frecvenţa noilor tag-uri



Figura 3.5 – Distribuţia frecvenţei noilor tag-uri (%)

Se observa ca aproape un sfert de cuvinte din setul de date sunt substantive, deci daca am creea un etichetator cu parametru implicit “substantiv”, am putea sa predictionam corect 23.56% de cuvinte din setul de date.

**Implementare part of speech classifier**

Procedeul principal este de a itera vechea lista cu cuvinte si tag-uri si de a creea/schimba lista in care vechile tag-uri vor face parte dintr-una din cele 10 categorii.

**foreach** **(**var w **in** Words**)**

**{**

int tagIndex **=** GetTagIndexForConversion**(**w**);**

string newTag **=** ConvertBrownTagToHierarchicTag**(**tagIndex**);**

newWords**.**Add**(new** Tokenizer**.**WordTag**(**w**.**word**,** newTag**));**

**}**

Pentru a gasi indexul la tag in lista avem urmatorul cod:

int tagIndex **=** **-**1**;**

**for** **(**int i **=** 0**;** i **<** BrownCorpusTags**.**Count**;** i**++)**

**{**

string**[]** splittedTag **=** Word**.**tag**.**Split**(new** Char**[]** **{** '+'**,** '-' **});**

**foreach** **(**string w **in** splittedTag**)**

**{**

**if** **(**Word**.**tag**.**Equals**(**"wql"**)** **||** Word**.**tag**.**Equals**(**"wql-tl"**))**

**{**

tagIndex **=** 55**;**

**return** tagIndex**;**

**}**

**else** **if** **(**Word**.**tag**.**Contains**(**BrownCorpusTags**[**i**]))**

**{**

tagIndex **=** i**;**

**return** tagIndex**;**

**}**

**}**

**}**

**return** tagIndex**;**

Aceasta returneaza un index in functie de unde se afla tag-ul in lista BrownCorpusTags, mai intai va desparti tag-urile compuse si va itera fiecare tag in parte pentru a ii atribuii un index in lista. Se va verifica daca exista tag-ul iterat in lista BrownCorpusTags, daca exista se va returna index-ul din lista, daca nu, se va returna -1. Metoda cauta tag-ul cu functia Contains deoarece unele tag-uri precum “nps$” apar si cu simboluri dar apartin tot in aceeasi categorie unde “nps” catalogat.

Lista BrownCorpusTags contine urmatoarele tag-uri:

"nn"**,** "nns"**,** "nns$"**,** "np"**,** "np$"**,** "nps"**,** "nps$"**,** "nr"**,** "nrs"**,**

"pn"**,** "pn$"**,** "pp$"**,** "pp$$"**,** "ppl"**,** "ppls"**,** "ppo"**,** "pps"**,** "ppss"**,** "wp$"**,** "wpo"**,** "wps"**,**

"vb"**,** "vbd"**,** "vbg"**,** "vbn"**,** "vbz"**,** "bem"**,** "ber"**,** "bez"**,** "bed"**,** "bedz"**,** "ben"**,** "do"**,** "dod"**,** "doz"**,** "hv"**,**

"hvd"**,** "hvg"**,** "hvn"**,** "hvz"**,** "md"**,**

"jj"**,** "jjr"**,** "jjs"**,** "jjt"**,**

"rb"**,** "rbr"**,** "rbt"**,** "rn"**,** "rp"**,** "wrb"**,** "ql"**,** "qlp"**,**

"in"**,** "to"**,**

"cc"**,** "cs"**,**

"at"**,** "ap"**,** "abl"**,** "abn"**,** "abx"**,** "dt"**,** "dti"**,** "dts"**,** "dtx"**,** "be"**,** "beg"**,** "ex"**,** "wdt"**,**

"."

, tag-urile din aceasta lista sunt ordonate in functie de categoriile descrise anterior. Conditia evidentiata din functie este un caz special deoarece gaseste secventa cu pattern-ul “ql” inainte de a gasi “wql” in lista(“ql” l-am catalogat ca adverb iar daca va cauta cu functia Contains atunci “wql” va fi catalogat ca adverb cand ar trebuii sa fie conjunctie) si de accea se verifica mai intai daca daca tag-ul este egal cu “wql” sau “wql-tl” (aici nu apar tag-uri compuse cu ‘+’ sau cu tag-uri “fw”, “hl” etc.). Tag-urile care nu apar in lista vor avea index = -1, si vor fi catalogate ca tag de Others (altele).

Conditia de clasificare ConvertBrownTagToHierarchicTag:

string tag **=** "Tag NOT found! Something went wrong!"**;**

**if** **(**tagIndex **>=** 0 **&&** tagIndex **<=** 8**)**

tag **=** "NN"**;**

**else** **if** **(**tagIndex **>=** 9 **&&** tagIndex **<=** 20**)**

tag **=** "PN"**;**

**else** **if** **(**tagIndex **>=** 21 **&&** tagIndex **<=** 40**)**

tag **=** "VB"**;**

**else** **if** **(**tagIndex **>=** 41 **&&** tagIndex **<=** 44**)**

tag **=** "JJ"**;**

**else** **if** **(**tagIndex **>=** 45 **&&** tagIndex **<=** 52**)**

tag **=** "RB"**;**

**else** **if** **(**tagIndex **>=** 53 **&&** tagIndex **<=** 54**)**

tag **=** "PP"**;**

**else** **if** **(**tagIndex **>=** 55 **&&** tagIndex **<=** 56**)**

tag **=** "CC"**;**

**else** **if** **(**tagIndex **>=** 57 **&&** tagIndex **<=** 69**)**

tag **=** "AT/DT"**;**

**else** **if** **(**tagIndex **==** 70**)**

tag **=** "."**;**

**else**

tag **=** "OT"**;**

**return** tag**;**

Aceasta este metoda care, in functie de index-ul returnat de metoda anterioara, va decide noul tag pentru cuvant.

ex. pentru setul: exemplu = (“cuvant\_exemplu”, “nps$”),

GetTagIndexForConversion va returna 6,

ConvertBrownTagToHierarchicTag va returna “NN”

Setul nou: exemplu 🡺 (“cuvant\_exemplu”, “NN”)

#### 3.2.2.3 Data cleaning & normalization

Acest procedeu presupune trecerea fiecarui cuvant printr-un filtru de preprocesare pentru a elimina cuvintele nepotrivite pentru algoritm si pentru a normaliza datele astfel incat sa nu incurce algoritmul de invatare.

Algoritmul de preprocesare va elimina mai intai cuvintele (caracterele) de oprire (caracterele care nu sunt importante la etichetarea partii de vorbire), acestea fiind: parantezele rotunde ‘()’, parantezele patrate ‘[]’ si acoladele ‘{}’. Dupa aceasta etapa, algoritmul verifica daca cuvantul procesat este un numar, daca contine doar cifre atunci il elimina, daca contine si cifre dar si litere atunci elimina cifrele din acesta si daca trece un anume prag de litere ramase atunci il pastreaza. Inainte de a termina, acesta in etapa de antrenare, va pastra o lista de cuvinte care incep doar cu litera mare si inca o lista in care toate cuvintele sunt normalizate la litera mica.

Cuvintele de testare vor trece prin aceelasi filtru, cu exceptia ultimei etape, acestea vor ramane neschimbate, nu se vor trece doar la litera mare si nu vor fi normalizate la litera mica.

Acest procedeu de normalizare va ajuta mai tarziu algoritmul de predictie, deoarece va putea sa faca distinctia corecta intre partea de vorbire a cuvintelor cu litera mare si cele cu litera mica. In setul de testare, de asemenea, se vor elimina duplicate(repetitiile) la caracterele cu tag de final de propozitie/fraza, deoarece acestea nu sunt evaluate de predictor si repetitiile de genul “….” pot influenta negativ acuratetea sistemului daca nu sunt tratate.

Clasa de normalizare si de curatare din clasa de preprocesare, contin si metode care vor fi utilizate mai tarziu de model si de decodor.

**Implementare Data cleaning & normalization**

Implementare preprocessing pipeline:

**public** static List**<**Tokenizer**.**WordTag**>** PreProcessingPipeline**(**

List**<**Tokenizer**.**WordTag**>** words**,**

bool toLowerOption **=** **false,**

bool keepOnlyCapitalizedWords **=** **false)**

**{**

List**<**Tokenizer**.**WordTag**>** newWords **=** **new** List**<**Tokenizer**.**WordTag**>();**

**foreach** **(**var sw **in** words**)**

**{**

**if** **(**Cleaning**.**IsStopWord**(**sw**.**word**))** **continue;**

string tsw **=** Cleaning**.**EliminateDigitsFromWord**(**sw**.**word**);**

**if** **(**string**.**IsNullOrEmpty**(**tsw**))** **continue;**

**if** **(**toLowerOption**)**

tsw **=** Normalization**.**ToLowerCaseNormalization**(**tsw**);**

**if(**keepOnlyCapitalizedWords**)**

**if** **(!**char**.**IsUpper**(**tsw**[**0**]))**

**continue;**

newWords**.**Add**(new** Tokenizer**.**WordTag**(**tsw**,** sw**.**tag**));**

**}**

**return** newWords**;**

**}**

Implementare functia care elimina duplicatele pentru tag-urile EOS (end of sentence):

**public** static void EliminateAllEndOfSentenceTags**(**

**ref** List**<**Tokenizer**.**WordTag**>** testWords**)**

**{**

testWords**.**RemoveAll**(**x **=>** x**.**tag **==** "."**);**

**}**

Implementare functie de eliminare cifre:

**public** static string EliminateDigitsFromWord**(**string word**)**

**{**

**if** **(!**word**.**Any**(**char**.**IsDigit**))**

**return** word**;**

**else**

**{**

string output **=** Regex**.**Replace**(**word**,** @"[\d-]"**,** string**.**Empty**);**

var count **=** output**.**Count**(**char**.**IsLetter**);**

const int x **=** 3**;**

**if** **(**count **>=** x**)** // verifies if has at least x letters left

**return** output**;**

**return** string**.**Empty**;**

**}**

**}**

Folosind sintaxa de Regex, in sectiunea evidentiata, acesta inlocuieste locul unde apare o cifra(d = digit) cu un caracter gol (adica elimina cifra). Functia ToLowerCaseNormalization(string) este echivalenta cu string.ToLower().

### 3.2.3 Model

Modelul este partea principala din sistem, acesta contine diferite informatii despre ponderile si valorile statistice calculate pe baza datelor antrenate. El este format din 2 submodele:

* Hidden Markov model
* Unknown words model

#### 3.2.3.1 Hidden Markov Model

Hidden Markov Model (HMM) este un model stochastic (foloseste metode probabilistice pentru a rezolva probleme) si este implementat dupa modelul clasic prezentat. Acesta foloseste probabilitatile de emisie si de tranzitie interpolate, pentru a predictiona tagul cuvintelor dintr-o propozitie.

##### Emission probability

Probabilitatea de emisie (sau likelihood) se calculeaza dat fiind un tag, care este probabilitatea de asociere al acestuia cu un cuvant dat.

|  |  |
| --- | --- |
|  | (3.2) |
| Unde, probabilitatea de emisie a cuvantului luat impreuna cu tagul  frecventa de aparitie a cuvantului luat impreuna cu tagul  frecventa de aparitie a tagului  Aceasta probabilitate este calculata pentru setul de testare dar numaratorile sunt facute in setul de antrenare. | | |
| ex. … ***he/???***went/VB home/NN … | (3.3) |

Putem vedea in exemplul anterior cum aceasta formula este o generalizare de la formula naiva a lui Bayes, important de remarcat ca aceasta nu calculeaza “care este cel mai probabil tag pentru cuvantul ‘he’?, asta s-ar calcula ”, ci “daca am genera un pronume, cat de probabil ar fi acesta ‘he’?’ (Jurafsky slp3-post, 2019). Valorile pentru frecventa de aparitie a cuvantului “he” cu pronume si frecventa de aparitie totala a tag-ului de pronume sunt alese aleator.

**Implementare emission probability**

Pentru a putea sa aplicam formula de la (3.2), mai intai trebuie sa obtinem numaratoarea cu aparitia tuturor tag-urilor din setul de antrenare iar dupa aceea sa obtinem aparitia tuturor cuvintelor din acest set impreuna cu fiecare tag care este intalnit cu acesta. Pentru a pastra fiecare aparitie separat, se va creea o lista de tip EmissionModel:

**public** class EmissionModel

**{**

**public** string Word**;**

**public** Dictionary**<**string**,** int**>** TagFreq**;**

**public** EmissionModel**()**

**{**

**this.**TagFreq **=** **new** Dictionary**<**string**,** int**>();**

**}**

**public** EmissionModel**(**string Word**,** Dictionary**<**string**,** int**>** TagFreq**)**

**{**

**this.**Word **=** Word**;**

**this.**TagFreq **=** TagFreq**;**

**}**

**}**

Cu aceasta putem sa tinem separat cuvantul si un dictionar de contorizare pentru fiecare tag.

Mentionat la capitolul “Data cleaning & normalization”, aici aplica algoritmul separat pentru cuvintele cu litere mari si toate cuvintele cu litera mica. Codul pentru cuvintele cu litere mari:

**this.**WordCapitalizedTagsEmissionFrequence **=** **new** List**<**EmissionModel**>();**

**foreach** **(**var w **in** capitalizedWords**)**

**{**

EmissionModel wmFind **=** WordCapitalizedTagsEmissionFrequence**.**

Find**(**x **=>** x**.**Word **==** w**.**word**);**

**if** **(**wmFind **==** **null)**

**{**

EmissionModel wModel **=** **new** EmissionModel**();**

wModel**.**Word **=** w**.**word**;**

wModel**.**TagFreq**.**Add**(**w**.**tag**,** 1**);**

**this.**WordCapitalizedTagsEmissionFrequence**.**Add**(**wModel**);**

**}**

**else**

**{**

var tag **=** wmFind**.**TagFreq**.**FirstOrDefault**(**x **=>** x**.**Key **==** w**.**tag**);**

**if** **(**tag**.**Key **==** **null)**

**{**

wmFind**.**TagFreq**.**Add**(**w**.**tag**,** 1**);**

**}**

**else**

**{**

wmFind**.**TagFreq**[**tag**.**Key**]** **+=** 1**;**

**}**

**}**

**}**

Se aplica sintaxa LINQ [4], pentru a cauta daca in lista cu frecvente avem deja cuvantul pe care il iteram (.Find(x => x.Word == w.word)), daca nu gasim cuvantul in lista atunci creem un obiect de tip EmissionModel si adaugam cuvantul si tag-ul aferent acestuia cu un count = 1, dupa care il adaugam in lista WordCapitalizedTagsEmissionFrequence. Daca totusi gasim cuvantul in lista, atunci cautam tag-ul aferent acestuia (metoda LINQ .FristOrDefault, va returna primul obiect din lista/dictionarul cautat, daca il va gasi, daca nu, va returna null), daca nu gasim tag-ul atunci adaugam noul tag la dictionar, altfel daca acel tag pe care il cautam exista, atunci incrementam count-ul tag-ului de la cheia unde l-am gasit. Aceeasi procedura si pentru cuvintele cu litera mica:

**this.**WordTagsEmissionFrequence **=** **new** List**<**EmissionModel**>();**

**foreach** **(**var w **in** uncapitalizedWords**)**

**{**

EmissionModel wmFind **=** WordTagsEmissionFrequence**.**

Find**(**x **=>** x**.**Word **==** w**.**word**);**

**if** **(**wmFind **==** **null)**

**{**

EmissionModel wModel **=** **new** EmissionModel**();**

wModel**.**Word **=** w**.**word**;**

wModel**.**TagFreq**.**Add**(**w**.**tag**,** 1**);**

**this.**AddTagToUnigramOccurences**(**w**.**tag**);**

**this.**WordTagsEmissionFrequence**.**Add**(**wModel**);**

**}**

**else**

**{**

var tag **=** wmFind**.**TagFreq**.**FirstOrDefault**(**x **=>** x**.**Key **==** w**.**tag**);**

**if** **(**tag**.**Key **==** **null)**

**{**

**this.**AddTagToUnigramOccurences**(**w**.**tag**);**

wmFind**.**TagFreq**.**Add**(**w**.**tag**,** 1**);**

**}**

**else**

**{**

**this.**AddTagToUnigramOccurences**(**w**.**tag**);**

wmFind**.**TagFreq**[**tag**.**Key**]** **+=** 1**;**

**}**

**}**

**}**

Odata ce am calculat frecventa de aparitie a tuturor tag-ului fiecarui cuvant, putem sa aplicam formula (3.2) pentru cuvintele care le gasim in setul de testare. De remarcat ca nu calculam probabilitatea tututoror cuvintelor din setul de antrenare ci doar acelora pe care le vom folosi in continuare, deci functia care calculeaza probabilitatea de emisie se va aplica doar pe setul de testare. Pentru a putea, din nou, retine informatii despre acestea vom avea nevoie de un dictionar care va retine probabilitati (tip double/float) in loc de contorizari:

**public** class EmissionProbabilisticModel

**{**

**public** string Word**;**

**public** Dictionary**<**string**,** double**>** TagFreq**;**

**public** EmissionProbabilisticModel**()**

**{**

**this.**TagFreq **=** **new** Dictionary**<**string**,** double**>();**

**}**

**public** EmissionProbabilisticModel**(**string Word**,**

Dictionary**<**string**,** double**>** TagFreq**)**

**{**

**this.**Word **=** Word**;**

**this.**TagFreq **=** TagFreq**;**

**}**

**}**

Codul de implementare a probabilitatii de emisie pentru toate cuvintele cu litera mica:

**foreach** **(**var tw **in** testWords**)**

**{**

string sWord **=** tw**.**word**.**ToLower**();**

PartOfSpeechModel**.**EmissionModel wmFind **=** WordTagsEmissionFrequence**.**

Find**(**x **=>** x**.**Word **==** sWord**);**

EmissionProbabilisticModel wFind **=** WordTagsEmissionProbabilities**.**

Find**(**x **=>** x**.**Word **==** sWord**);**

**if** **(**wmFind **!=** **null** **&&** wFind **==** **null)**

**{**

EmissionProbabilisticModel epModel **=** **new** EmissionProbabilisticModel**();**

epModel**.**Word **=** wmFind**.**Word**;**

**foreach** **(**var tf **in** wmFind**.**TagFreq**)**

**{**

int cti **=** **this.**UnigramFrequence**.**

FirstOrDefault**(**x **=>** x**.**Key **==** tf**.**Key**).**Value**;**

double pwiti **=** **(**double**)**tf**.**Value **/** cti**;** // Emission probability: p(wi/ti) = C(ti, wi) / C(ti)

epModel**.**TagFreq**.**Add**(**tf**.**Key**,** pwiti**);**

**}**

**this.**WordTagsEmissionProbabilities**.**Add**(**epModel**);**

**}**

**}**

In conditia evidentiata, verificam daca gasim mai intai cuvantul de testat in lista antrenata si daca cuvantul de testat nu este deja in lista procesata cu probabilitatile de emisie. Daca cuvantul apare pentru prima data si il gasim in lista de antrenare, atunci putem continua prin a itera toate tag-urile cuvantului, si de a calcula probabilitatea de emisie. Variabila cti va cauta tag-ul respectiv in lista cu frecventele unigramului (aceasta fiind descrisa la sectiunea “Bigram & unigram”), dupa care putem imparti deimpartitorul tf.Value, care este contorizarea tag-ului actual asociat cuvantului de testare actual, supra frecventa totala a tag-ului respectiv.

Pentru cuvintele cu litera mare se va aplica aceelasi proces doar ca rezultatele probabilitatii se vor pastra intr-o lista diferita(WordCapitalizedTagsEmissionProbabilities) si in loc de prima linie din foreach:

string sWord **=** tw**.**word**.**ToLower**();**

, se va verifica daca primul caracter este cu litera mare:

**if** **(!**char**.**IsUpper**(**tw**.**word**[**0**]))** **continue;**

##### Transition probability

Probabilitatea de tranzitie (sau prior probability) se calculeaza dat fiind un tag, care este probabilitatea de aparitie dupa un anume tag dat.

|  |  |
| --- | --- |
|  | (3.4) |

Unde, probabilitatea bigramului pentru secventa de tag-uri

frecventa de aparitie a secventei bigram a tag-urilor

frecventa de aparitie a tag-ului

ex. … Anna/NN ***likes/VB???*** ice-cream/JJ …

|  |  |
| --- | --- |
|  | (3.5) |

In exemplul dat, calculam probabilitatea ca “likes” luat ca verb, sa urmeze dupa un substantiv, adica numarul de aparitii a unui substantiv (NN) urmat de un verb (VB) este 8027 supra numarul de aparitii total a unui substantiv (NN) este 13038, rezultatul fiind 0.61 in exemplul de sus.

**Bigram & unigram**

Formula data la (3.4) este formula de calculare a probabilitatii unui bigram (sau 2-gram), deoarece este formata din numarul de aparitii a 2 tag-uri urmat unul dupa altul. Daca am avea doar un singur tag atunci se va numi unigram (1-gram) si are formula urmatoare:

|  |  |
| --- | --- |
|  | (3.6) |

Unde, probabilitatea unigramului pentru tag-ul

frecventa de aparitie a tag-ului

N – numarul de tokeni (cuvinte/caractere) din setul de antrenare

Cu aceste formulele (3.2), (3.4) si (3.6) putem creea un sistem de predictie bazat pe HMM dar doar cu acestea nu vom obtine cele mai bune rezultate.

**Implementare Bigram & unigram**

Pentru a putea sa implementam formulele de la (3.5) si (3.6) mai intai va trebuii sa obtinem frecventele de aparitie a tag-ului individual si a 2 tag-uri unul dupa altul. Pentru asta avem nevoie de un dictionar care sa tina o evidenta a tuturor tranzitiilor:

**private** Dictionary**<**string**,** int**>** UnigramFrequence;

**private** Dictionary**<**Tuple**<**string**,** string**>,** int**>** BigramTransitionFrequence**;**

Codul pentru a numara secventele de bigram din setul de antrenare:

**this.**BigramTransitionFrequence **=** **new** Dictionary**<**Tuple**<**string**,** string**>,** int**>();**

bool firstFileChecked **=** **false;**

**for** **(**int i **=** **-**1**;** i **<** wordsInput**.**Count **-** 1**;** i**++)**

**{**

**if** **(!**firstFileChecked**)**

**{**

**this.**BigramTransitionFrequence**.**Add**(**

**new** Tuple**<**string**,** string**>(**"."**,** wordsInput**[**i **+** 1**].**tag**),** 1**);**

firstFileChecked **=** **true;**

**continue;**

**}**

var tuple **=** **new** Tuple**<**string**,** string**>(**wordsInput**[**i**].**tag**,**

wordsInput**[**i **+** 1**].**tag**);**

var tag **=** **this.**BigramTransitionFrequence**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**tuple**));**

**if** **(**tag**.**Key **==** **null)**

**{**

**this.**BigramTransitionFrequence**.**Add**(**tuple**,** 1**);**

**}**

**else**

**{**

**this.**BigramTransitionFrequence**[**tag**.**Key**]** **+=** 1**;**

**}**

**}**

Se poate observa in conditia evidentiata ca s-a pus un flag de verificare, in acesta se intra la prima iteratie a for-ului. Se realizeaza aceasta verificare pentru prima tranzitie din setul de antrenare, (NULL, NN) este echivalenta cu (“.”, NN) deoarece inceput de propozitie este o tranzitie valida care ne spune cu ce tag incepe propozitia. Cum primul token din setul de antrenare nu are un flux de date inaintea lui atunci va trebuii sa ii se faca aceasta verificare. Pentru a pastra tranzitiile intre partile de vorbire, s-a ales ca tip de date un tuplu cu 2 parametrii, primul parametru fiind cuvantul de dinainte (precedentul) si al doilea parametru fiind cuvantul care urmeaza dupa precedent. Implementarea este foarte asemanatoare la concept cu implementarea din probabilitatea de emisie.

Pentru unigram, implementarea este urmatoarea:

**private** void AddTagToUnigramOccurences**(**string wordTag**)**

**{**

var tag **=** **this.**UnigramFrequence**.**FirstOrDefault**(**x **=>** x**.**Key **==** wordTag**);**

**if** **(**tag**.**Key **==** **null)**

**{**

**this.**UnigramFrequence**.**Add**(**wordTag**,** 1**);**

**}**

**else**

**{**

**this.**UnigramFrequence**[**tag**.**Key**]** **+=** 1**;**

**}**

**}**

Vazusem anterior la sectiunea “Emission probability” utilizarea acestei metode AddTagToUnigramOccurences(..). In loc sa iteram de 2 ori setul de antrenare, il iteram doar o singura data si adaugam in aceelas timp si frecventele cuvintelor cu tag-uri (word-tag emission count) si frecventele tuturor tag-urilor individuale (unigram count).

Diferenta fata de calcularea probabilitatii la lista cu emisie si dictionarele cu unigram si bigram, este aceea ca tuplurile din dictionarele cu unigram si bigram se vor calcula indiferent deoarece acestea sunt “arcele(graf orientat)” **starilor ascunse** si nu apar in setul de testare. Pentru a putea stii cate combinatii totale putem avea in total la un n-gram, se da urmatoarea formula:

|  |  |
| --- | --- |
|  | (3.7) |

unde, T – numarul total de combinatii posibile intr-un n-gram

x – numarul tuturor partilor de vorbire individuale din corpus

n – n-gram-ul ales (2 in cazul bigram, 3 in cazul trigram etc.)

ex. avem n = 2 (2-gram) si x = 10 (10 tag-uri diferite),

|  |  |
| --- | --- |
|  | (3.8) |

Intr-un bigram cu 10 parti de vorbire diferite, putem avea in total 10 stari cu 100 de arce.

Functiile de calculare a unigram-ului si a bigram-ului sunt urmatoarele:

**private** void calculateUnigramTestCorpus**()**

**{**

**foreach** **(**var uni **in** **this.**UnigramFrequence**)**

**{**

double pi **=** **(**double**)(**uni**.**Value **-** 1**)** **/** **(this.**N **-** 1**);**

**this.**UnigramProbabilities**.**Add**(**uni**.**Key**,** pi**);**

**}**

**}**

**private** void calculateBigramTestCorpus**()**

**{**

**foreach** **(**var bi **in** **this.**BigramTransitionFrequence**)**

**{**

var cti **=** **this.**UnigramFrequence**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**bi**.**Key**.**Item1**)).**Value**;**

double pti **=** **(**double**)(**bi**.**Value **-** 1**)** **/** **(**cti **-** 1**);** // Transition probability: p(ti|ti-1) = C(ti-1, ti) / C(ti-1)

**this.**BigramTransitionProbabilities**.**Add**(**bi**.**Key**,** pti**);**

**}**

**}**

Calculul probabilitatilor de tranzitie au fost scazute cu o constanta = 1 si la numarator si la numitor, in sectiunea viitoare “Trigram & bigram smoothing” se explicita motivul pentru aceasta decizie. Rezultatele vor fi puse in dictionare unde valoarea rezultate va fi una de tip double (probabilitate):

**public** Dictionary**<**string**,** double**>** UnigramProbabilities**;**

**public** Dictionary**<**Tuple**<**string**,** string**>,** double**>**BigramTransitionProbabilities**;**

**Trigram**

In multe cazuri, uitandu-ne doar la cuvantul de dinainte nu e destul, avem nevoie de mai multa informatie, pentru expresiile compuse formate din 3 cuvinte, stiind intreg contextul e mai util decat sa cunoastem doar cuvantul de dinainte. Pentru asta se introduce trigram-ul, acesta se uita inapoi la 2 cuvinte fata de bigram care se uita doar la cuvantul de dinainte. Pentru a calcula probabilitatea trigramului se da formula:

|  |  |
| --- | --- |
|  | (3.9) |

Unde, probabilitatea trigramului a secventei de tag-uri

frecventa de aparitie a secventei trigram ale tag-urilor

frecventa de aparitie a secventei bigram a tag-urilor

ex. The/DT red/JJ ***hat/NN???*** is/VB …

|  |  |
| --- | --- |
|  | (3.10) |

In exemplul anterior, se calculeaza ca, dat fiind bigramul cu cele 2 tag-uri precedente (DT-determinant, JJ-adjectiv), care este probabilitatea de aparitie a unui substantiv (NN). Trigramul este cel mai potrivit n-gram pentru etichetarea partii de vorbire pentru un model de tip Hidden Markov, returneaza cea mai buna acuratete dintre cele trei (unigram,bigram,trigram) dar este si cel mai intensiv din punct de vedere al timpului computational.

**Implementare trigram**

Acesta va fi implementat tot intr-un tip de date dictionar cu un tuplu cu 3 parametrii:

Frecventa aparitii:

Dictionary**<**Tuple**<**string**,** string**,** string**>,** int**>** TrigramTransitionFrequence**;**

Probabilitati:

Algoritmul implementat de numarare a frecventelor de aparitie:

Dictionary**<**Tuple**<**string**,** string**,** string**>,** double**>** TrigramTransitionProbabilities**;**

**private** void CalculateTrigramOccurences**(**List**<**Tokenizer**.**WordTag**>** wordsInput**)**

**{**

**this.**TrigramTransitionFrequence **=** **new** Dictionary**<**Tuple**<**string**,**

string**,** string**>,** int**>();**

bool firstFileChecked **=** **false;**

**for** **(**int i **=** **-**1**;** i **<** wordsInput**.**Count **-** 2**;** i**++)**

**{**

**if** **(!**firstFileChecked**)**

**{**

**this.**TrigramTransitionFrequence**.**Add**(new** Tuple**<**string**,**

string**,** string**>(**"."**,**

wordsInput**[**i **+** 1**].**tag**,**

wordsInput**[**i **+** 2**].**tag**),** 1**);**

firstFileChecked **=** **true;**

**continue;**

**}**

var tuple **=** **new** Tuple**<**string**,** string**,** string**>(**wordsInput**[**i**].**tag**,**

wordsInput**[**i **+** 1**].**tag**,**

wordsInput**[**i **+** 2**].**tag**);**

**if** **(**tuple**.**Item2**.**Equals**(**"."**))**

**continue;**

var tag **=** **this.**TrigramTransitionFrequence**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**tuple**));**

**if** **(**tag**.**Key **==** **null)**

**{**

**this.**TrigramTransitionFrequence**.**Add**(**tuple**,** 1**);**

**}**

**else**

**{**

**this.**TrigramTransitionFrequence**[**tag**.**Key**]** **+=** 1**;**

**}**

**}**

**}**

Algoritmul este foarte asemanator cu cel descris la bigram, diferenta majora fiind evidentiata in a doua conditie, aici nu mai salvam si secventele unde intalnim sfarsit de propozitie(sau inceput de propozitie) in mijlocul tuplului. Acest lucru este realizat deoarece, propozitiile/frazele sunt de sine statatoare, partea de vorbire a unui token de inceput nu depinde de tag-ul ultimului token din propozitia anterioara. Chiar daca asta ar putea ajuta sistemul la o acuratete mai buna (multe propozitii se termina cu un substantiv/verb si incep cu un substantiv sau articol), nu ar fi util in aplicatiile reale, deoarece de multe ori datele reale sunt introduse de userii care folosesc aplicatia (intr-o cantitate mult mai mica) ci nu sunt preluate dintr-un set mare de date.

Functia de calcul a probabilitatii trigram-ului:

**private** void calculateTrigramTestCorpus**()**

**{**

**foreach** **(**var tri **in** **this.**TrigramTransitionFrequence**)**

**{**

Tuple**<**string**,** string**>** tuple **=** **new** Tuple**<**string**,** string**>**

**(**tri**.**Key**.**Item1**,** tri**.**Key**.**Item2**);**

var cti **=** **this.**BigramTransitionFrequence**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**tuple**)).**Value**;**

double pti **=** **(**double**)(**tri**.**Value **-** 1**)** **/** **(**cti **-** 1**);** // Transition probability: p(ti|ti-1, ti-2) = C(ti-2, ti-1, ti) / C(ti-2, ti-1)

**this.**TrigramTransitionProbabilities**.**Add**(**tri**.**Key**,** pti**);**

**}**

**}**

**Trigram & bigram smoothing**

Probabilitatile de tip trigram generate dintr-un corpus *deobicei* nu pot fi folosite direct din cauza valorilor lipsa a tuplurilor trigram (sparse-data problem [2]). Pentru a rezolva aceasta problema se introduce conceptul de interpolare liniara. Acesta presupune calcularea a unei noi probabilitati compuse din suma probabilitatilor de tranzitie cu o pondere pentru fiecare:

|  |  |
| --- | --- |
|  | (3.11) |
|  |  |

Valorile sunt estimate prin interpolarea eliminata. Pseudocodul general de determinare a acestor ponderi este descris astfel:



Figura 3.6 – pseudocodul pentru interpolarea liniara (Deleted Interpolation) a trigram-ului [2]

Se poate observa cum probabilitatile fiecarui n-gram sunt scazute cu 1 si la numarator si la numitor, asta inseamna ca algoritmul ia in calcul date care nu au aparut in setul de antrenare. Cu toate acestea mentionate, interpolarea liniara nu este folositoare numai pentru a trata tupluri lipsa dar si pentru a seta ponderi fiecarui n-gram separat. Folosind aceasta functie, acuratetea predictorului creste, in cazul meu, cresterea acuratatii este mica deoarece setul de antrenare este clasificat in 10 clase (tag-uri) reprezentative si de aceea dictionarele bigram si trigram au aproape toate permutarile posibile. Folosind formula de la (3.7) se poate verifica n-gram-ul contine toate permutarile sale posibile, astfel tragand concluzia daca interpolarea liniara aduce imbunatatiri majore la predictie.

Inafara de interpolarea pentru trigram, s-a realizat si o interpolare pe bigram, aceasta este foarte asemanatoare cu cea de la trigram, ea este prezentata mai departe la implemetare.

**Implementare deleted interpolation**

Algoritmu interpolarii liniare pentru trigram:

**private** void DeletedInterpolationTrigram**()**

**{**

**if** **(this.**TrigramTransitionProbabilities **==** **null)**

**this.**TrigramTransitionProbabilities **=** **new** Dictionary**<**Tuple**<**string**,**

string**,** string**>,** double**>();**

int lambda1 **=** 0**,** lambda2 **=** 0**,** lambda3 **=** 0**;**

**foreach** **(**var tri **in** **this.**TrigramTransitionFrequence**)**

**{**

string unituple **=** tri**.**Key**.**Item3**;**

Tuple**<**string**,** string**>** bituple **=** **new** Tuple**<**string**,** string**>(**tri**.**Key**.**Item2**,**

tri**.**Key**.**Item3**);**

double univalue **=** **this.**UnigramProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**unituple**)).**Value**;**

double bivalue **=** **this.**BigramTransitionProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**bituple**)).**Value**;**

double trivalue **=** **this.**TrigramTransitionProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**tri**.**Key**)).**Value**;**

**if(**bivalue **<** univalue **&&** univalue **>** trivalue**)**

**{**

lambda1 **+=** tri**.**Value**;**

**}**

**else** **if(**univalue **<** bivalue **&&** bivalue **>** trivalue**)**

**{**

lambda2 **+=** tri**.**Value**;**

**}**

**else**

**{**

lambda3 **+=** tri**.**Value**;**

**}**

**}**

int sum **=** lambda1 **+** lambda2 **+** lambda3**;**

**this.**TgramLambda1 **=** **(**double**)**lambda1 **/** sum**;**

**this.**TgramLambda2 **=** **(**double**)**lambda2 **/** sum**;**

**this.**TgramLambda3 **=** **(**double**)**lambda3 **/** sum**;**

**}**

Se poate observa ca valorile sunt luate din dictionarele aferente fiecarui n-gram si nu sunt recalculate pe loc, pentru a putea sa scadem 1 si la numitor si la numarator, am realizat aceasta operatie la calcularea finala probabilitatii fiecarei tranzitii in sub-capitolul anterior. Valorile ponderilor lambda se vor pastra in membrii publici al clasei Model, fiind accesibile mai tarziu decodorului pentru a putea aplica formula de la (3.11).

Algoritmul pentru bigram:

**foreach** **(**var bi **in** **this.**BigramTransitionFrequence**)**

**{**

string unituple **=** bi**.**Key**.**Item2**;**

double univalue **=** **this.**UnigramProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**unituple**)).**Value**;**

double bivalue **=** **this.**BigramTransitionProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**bi**.**Key**)).**Value**;**

**if** **(**bivalue **<** univalue**)**

**{**

lambda1 **+=** bi**.**Value**;**

**}**

**else**

**{**

lambda2 **+=** bi**.**Value**;**

**}**

**}**

int sum **=** lambda1 **+** lambda2**;**

**this.**BgramLambda1 **=** **(**double**)**lambda1 **/** sum**;**

**this.**BgramLambda2 **=** **(**double**)**lambda2 **/** sum**;**

Dupa cum se poate vedea, algoritmul este aproape identic cu cel de la trigram, doar ca aici nu avem 3 ponderi ci doar 2, se itereaza valorile de bigram si acestea vor fi valorile adunate pentru ponderi. Valorile de ponderi vor fi si ele accesibile decodorului.

#### 3.2.2.1 Unknown words model

In sub-capitolul anterior, am prezentat modelul Markov, folosind doar acest model, sistemul de etichetare poate obtine o acuratete destul de ok dar un sistem bun de etichetare trebuie sa poata trata si cuvintele care nu se gasesc in setul de antrenare. Pentru asta exista mai multe modalitati de tratare precum: sistem bazat pe reguli, invatare nesupravegheata, algorimti care analizeaza structura cuvantantului etc. In aceasta lucrare, modelul pentru tratarea cuvintelor necunoscute este bazat pe doua parti, una de analiza a sufixelor/prefixelor cuvantului si o parte bazata pe reguli implementate manual in urma unei analize a setului de date. Functia finala care va combina aceste 2 parti, va primi 2 parametrii de intrare, acestea fiind cuvantul necunoscut si tag-ul cu care se crede ca este asociat iar la final, functia va returna o probabilitate a asocierii cuvantului cu tag-ul introdus. Pentru a putea folosi functia, mai intai va trebuii sa obtinem informatii pentru sufixe si prefixe de la setul de antrenare.

**Prefix & suffix training phase**

Prefixul este un afix ce este pus inaintea radacinii unui cuvant, iar sufixul este pus dupa radacina cuvantului. Aceste 2 componente importante pot da informatii despre partea de vorbire a unui cuvant, in jurnalul “Wall Street”, cuvintele care se termina cu “able” sunt adjective in 98% din cazuri [2].

ex. incompatibility (substantiv)

In exemplul acesta, “in” este prefixul axifului “compatibili” iar “ity” este sufixul acesteia. De remarcat aici ca si “ty” poate fi un sufix, atat cat si “y” (multe adverbe si adjective in engleza se termina cu caracterul “y”) dar sufixul intreg “ity” este cel mai specific pentru exemplul dat.

Pentru a putea alege cele mai bune sufixe/prefixe, acestea nu au fost deduse din setul de antrenament (timp computational mare si rezultate mediocre) ci au alese manual ca fiind cele mai reprezentative. In urma analizei din [5], [6], [7], prefixele si sufixele alese pentru sistem sunt:

List**<**string**>** pref **=** **new** List**<**string**>()** **{** "inter"**,** "intra"**,** "mis"**,** "mid"**,** "mini"**,** "dis"**,** "di"**,** "re"**,** "anti"**,** "in"**,** "en"**,** "em"**,** "auto"**,** "il"**,** "im"**,** "ir"**,** "ig"**,** "non"**,** "ob"**,** "op"**,** "octo"**,** "oc"**,** "pre"**,** "pro"**,** "under"**,** "epi"**,** "off"**,** "on"**,** "circum"**,** "multi"**,** "bio"**,** "bi"**,** "mono"**,** "demo"**,** "de"**,** "super"**,** "supra"**,** "cyber"**,** "fore"**,** "for"**,** "para"**,** "extra"**,** "extro"**,** "ex"**,** "hyper"**,** "hypo"**,** "hy"**,** "sub"**,**"com"**,** "counter"**,** "con"**,** "co"**,** "semi"**,** "vice"**,** "poly"**,** "trans"**,** "out"**,** "step"**,** "ben"**,** "with"**,** "an"**,** "el"**,** "ep"**,** "geo"**,** "iso"**,** "meta"**,** "ab"**,** "ad"**,** "ac"**,** "as"**,** "ante"**,** "pan"**,** "ped"**,** "peri"**,** "socio"**,** "sur"**,** "syn"**,** "sy"**,** "tri"**,** "uni"**,** "un"**,** "eu"**,** "ecto"**,** "mal"**,** "macro"**,** "micro"**,** "sus"**,** "ultra"**,** "omni"**,** "prim"**,** "sept"**,** "se"**,** "nano"**,** "tera"**,** "giga"**,** "kilo"**,** "cent"**,** "penta"**,** "tech"**};**

List**<**string**>** suff **=** **new** List**<**string**>()** **{** "able"**,** "ible"**,** "ble"**,** "ade"**,** "cian"**,** "ance"**,** "ite"**,** "genic"**,** "phile"**,** "ian"**,** "ery"**,** "ory"**,** "ary"**,** "ate"**,** "man"**,** "an"**,** "ency"**,** "eon"**,** "ex"**,** "ix"**,**"acy"**,** "escent"**,** "tial"**,** "cial"**,** "al"**,**

"ee"**,** "en"**,**"ence"**,** "ancy"**,** "eer"**,** "ier"**,** "er"**,** "or"**,** "ar"**,** "ium"**,** "ous"**,** "est"**,** "ment"**,** "ese"**,** "ness"**,** "ess"**,** "ship"**,** "ed"**,** "ant"**,** "ow"**,** "land"**,** "ure"**,** "ity"**,** "esis"**,** "osis"**,** "et"**,** "ette"**,** "ful"**,** "ify"**,** "ine"**,** "sion"**,** "fication"**,** "tion"**,** "ion"**,** "ish"**,** "ism"**,** "ist"**,** "ty"**,** "ly"**,** "em"**,** "fic"**,** "olve"**,** "ope"**,**

"ent"**,** "ise"**,** "ling"**,** "ing"**,** "ive"**,** "ic"**,** "ways"**,** "in"**,** "ology"**,** "hood"**,** "logy"**,** "ice"**,** "oid"**,** "id"**,** "ide"**,** "age"**,** "worthy"**,** "ae"**,** "es" **};**

Pentru putea folosi aceste afix-uri pentru a gasi tag-ul cuvintelor necunoscute, trebuie mai intai sa vedem daca apar in setul de antrenament si daca apar, cu ce probabilitate apar cu un tag specific.

Pentru a putea sa calculam acest lucru, se foloseste urmatoare formula:

|  |  |
| --- | --- |
|  | (3.12) |

Unde, –probabilitatea de asociere a unui sufix/prefix , cu tagul

– frecventa de aparitie a sufix-ului/prefix-ului cu tagul

– constanta pentru a realiza “additive smoothing” [8]

– suma tuturor tag-urilor asociate sufix-ului/prefix-ului

d – marimea totala a setului de prefixe/sufixe

Se observa ca reapare conceptul de smoothing, acesta este foarte important aici deoarece vrem sa obtinem o probabilitate diferita de 0 si pentru sufixele/prefixele care nu le gasim in setul de antrenament. In program, constanta a fost initiata cu , aceasta metoda este numita si **Laplace Smoothing**. [8]

ex.

|  |  |
| --- | --- |
|  | (3.13) |

Aceste calcule prezentate au fost realizate pentru sufixe/prefixe care incep cu litera mica dar in sistem este implementat si calculul pentru cuvinte care incep cu litera mare deoarece aceste distincii intre cuvinte pot da rezultate mai bune la evaluarea sistemului.

**Implementare prefix & suffix training phase**

Pentru a putea calcula probabilitatea de la (3.12), avem nevoie sa frecventele de aparitie a sufix-ului/prefix-ului cu fiecare tag. Algoritmul de contorizare a sufixelor si a prefixelor la cuvintele care incep cu litera mica:

**foreach** **(**var w **in** uncapitalizedWords**)**

**{**

**foreach** **(**var sfx **in** suffxem**)**

**{**

**if** **(**w**.**word**.**EndsWith**(**sfx**.**Word**))**

**{**

var tag **=** sfx**.**TagFreq**.**FirstOrDefault**(**x **=>** x**.**Key **==** w**.**tag**);**

**if** **(**tag**.**Key **==** **null)**

**{**

sfx**.**TagFreq**.**Add**(**w**.**tag**,** 1**);**

**}**

**else**

**{**

sfx**.**TagFreq**[**tag**.**Key**]** **+=** 1**;**

**}**

**}**

**}**

**foreach** **(**var pfx **in** preffxem**)**

**{**

**if** **(**w**.**word**.**StartsWith**(**pfx**.**Word**))**

**{**

var tag **=** pfx**.**TagFreq**.**FirstOrDefault**(**x **=>** x**.**Key **==** w**.**tag**);**

**if** **(**tag**.**Key **==** **null)**

**{**

pfx**.**TagFreq**.**Add**(**w**.**tag**,** 1**);**

**}**

**else**

**{**

pfx**.**TagFreq**[**tag**.**Key**]** **+=** 1**;**

**}**

**}**

**}**

**}**

Pentru a le adauga in lista de sufixe si de prefixe acestea folosesc tot un tip de date EmissionModel prezentat la capitolul “Emission probabilities”, string word fiind in acest caz sirul de caractere de tip sufix/prefix si TagFreq fiind tot lista de tag-uri. De remarcat ca ordinea sufixelor si a prefixelor, din lista implementarii lor manuale prezentata anterior, este foarte importanta, daca noi apelam metoda .EndsWith(..) si un sufix de tipul “ty” se afla inaintea la sufixul “ity” atunci, sufixul “ity” nu va mai putea fi contorizat in lista de sufixe. Acest proces este exact la fel pentru sufixele si prefixele cuvintelor care incep cu litera mare, doar ca pentru prefixe se trece cuvantul de la litera mare la litera mica cu metoda .ToLower().

Lista finala cu probabilitati va fi una de tip EmissionProbabilisticModel, algoritmul de calculare a probabilitatii (3.12) pentru cuvinte care incep cu litera mica este urmatorul:

**foreach** **(**var sfx **in** suffxem**)**

**{**

var tagSum **=** sfx**.**TagFreq**.**Sum**(**x **=>** x**.**Value**);**

Dictionary**<**string**,** double**>** tgfreq **=** **new** Dictionary**<**string**,** double**>();**

**foreach** **(**var tg **in** sfx**.**TagFreq**)**

**{**

tgfreq**.**Add**(**tg**.**Key**,** **(**double**)(**tg**.**Value **+** smoothing**)** **/**

**(**tagSum **+** **(**smoothing **\*** suffSize**)));**

**}**

var em **=** **new** EmissionProbabilisticModel**();**

em**.**Word **=** sfx**.**Word**;**

em**.**TagFreq **=** tgfreq**;**

**this.**SuffixEmissionProbabilities**.**Add**(**em**);**

**}**

**foreach** **(**var pfx **in** preffxem**)**

**{**

var tagSum **=** pfx**.**TagFreq**.**Sum**(**x **=>** x**.**Value**);**

Dictionary**<**string**,** double**>** tgfreq **=** **new** Dictionary**<**string**,** double**>();**

**foreach** **(**var tg **in** pfx**.**TagFreq**)**

**{**

tgfreq**.**Add**(**tg**.**Key**,** **(**double**)(**tg**.**Value **+** smoothing**)** **/**

**(**tagSum **+** **(**smoothing **\*** prefSize**)));**

**}**

var em **=** **new** EmissionProbabilisticModel**();**

em**.**Word **=** pfx**.**Word**;**

em**.**TagFreq **=** tgfreq**;**

**this.**PrefixEmissionProbabilities**.**Add**(**em**);**

**}**

Pentru cuvintele care incep cu litera mare procesul este exact la fel.

**Model training on separate threads**

In faza de creeare si de antrenare a modelului, se creaza listele importante cu informatii despre probabilitatile de emisie, de tranzitie si antrenarea sufixelor si a prefixelor. Calcularea acestora consuma timp daca sunt calculate una dupa cealalta, dar avand in vedere ca ele aceste componente sunt separate si nu influenteaza una pe cealalta, acestea pot fi calculate asincron pe threaduri diferite folosind clasa Task din .NET []. Implementarea fazei de antrenare a modelului este acesta:

**public** void CreateHiddenMarkovModel**(**

List**<**Tokenizer**.**WordTag**>** uncapitalizedWords**,**

List**<**Tokenizer**.**WordTag**>** capitalizedWords**,**

int smoothingCoef **=** 0**)**

**{**

**this.**N **=** uncapitalizedWords**.**Count**;**

// > .NET 4.0 for task-ing

Task taskSuffixPrefixEmission **=** Task**.**Factory**.**StartNew**(()** **=>**

**this.**GetEmissionProbabilitiesForSuffixesAndPrefixes**(**

uncapitalizedWords**,**

capitalizedWords**,**

smoothingCoef**));**

Task taskEmissionWords **=** Task**.**Factory**.**StartNew**(()** **=>**

**this.**CalculateEmissionForWordTags**(**

uncapitalizedWords**,**

capitalizedWords**));**

Task taskBigram **=** Task**.**Factory**.**StartNew**(()** **=>**

**this.**CalculateBigramOccurences**(**uncapitalizedWords**));**

Task taskTrigram **=** Task**.**Factory**.**StartNew**(()** **=>**

**this.**CalculateTrigramOccurences**(**uncapitalizedWords**));**

Task**.**WaitAll**(**taskSuffixPrefixEmission**,**

taskEmissionWords**,**

taskBigram**,**

taskTrigram**);**

**}**

Se declara un task pentru fiecare functie, acesta fiind lansat la apelarea functiei .StartNew(), iar la final se asteapta finalizarea tuturor pentru a putea iesi din metoda principala.

In faza de testare, cand se calculeaza probabilitatile reale de emisie si de tranzitie, se utilizeaza din nou clasa Task pentru a calcula in paralel aceste probabilitati:

**public** void CalculateHiddenMarkovModelProbabilitiesForTestCorpus**(**

List**<**Tokenizer**.**WordTag**>** testWords**,**

string model **=** "bigram"**)**

**{**

// emission stage

Task taskEmission **=** Task**.**Factory**.**StartNew**(()** **=>**

**this.**calculateEmissionTestCorpus**(**testWords**));**

// transition stage

// unigram

Task taskUnigram **=** Task**.**Factory**.**StartNew**(()** **=>**

**this.**calculateUnigramTestCorpus**());**

// bigram

Task taskBigram **=** Task**.**Factory**.**StartNew**(()** **=>**

**this.**calculateBigramTestCorpus**());**

**if** **(**model**.**Equals**(**"trigram"**))** // trigram

**{**

Task taskTrigram **=** Task**.**Factory**.**StartNew**(()** **=>**

**this.**calculateTrigramTestCorpus**());**

Task**.**WaitAll**(**taskEmission**,** taskUnigram**,** taskBigram**,** taskTrigram**);**

Task taskBiInterp **=** Task**.**Factory**.**StartNew**(()** **=>**

**this.**DeletedInterpolationBigram**());**

Task taskTriInterp **=** Task**.**Factory**.**StartNew**(()** **=>**

**this.**DeletedInterpolationTrigram**());**

Task**.**WaitAll**(**taskBiInterp**,** taskTriInterp**);**

**}**

**else**

**{**

Task**.**WaitAll**(**taskEmission**,** taskUnigram**,** taskBigram**);**

**this.**DeletedInterpolationBigram**();**

**}**

**}**

Folosind clasa Task din .NET, reducem timpul de antrenare a modelului deoarece functiile vor rula asincron pe fire diferite, in loc sa ruleze sincron pe firul principal unde rularea fiecarei functii se face dupa ce functia de dinainte termina de executat.

**Rule-based tag weights**

Ultima componenta din functia de recunoastere a tag-ului pentru cuvinte necunoscute ramane componenta bazata pe regulile impuse manual. Asta se bazeaza pe reguli precum, cuvintele care incep cu litera mare au o probabilitate mai mare sa fie substantive, cele cu apostrof si care se termina cu ‘s’ au o probabilitate foarte mare sa fie substantive, cele care contin cratima (‘-‘) au o probabilitate mai mare sa fie cuvinte compuse de tip OT (altele) sau JJ (adjectiv), etc. Pentru a seta la fiecare conditie o anumita pondere, s-au ales 2 parametrii care influenteaza ponderea finala pentru tag-ul care se verifica, aceste 2 valori sunt:

const double bestValueWeight **=** 2.5d**,** worstValueWeight **=** 1.5d**;**

Regulile impuse manual sunt:

bool testWordIsCapitalized **=** **false;**

**if** **(**char**.**IsUpper**(**testWord**[**0**]))**

testWordIsCapitalized **=** **true;**

string lowerWord **=** testWord**.**ToLower**();**

double occurenceAdder **=** 0.0d**;**

**if** **(**testWordIsCapitalized **&&** currentTag **==** "NN"**)**

occurenceAdder **+=** **(**double**)**bestValueWeight **/** 1.15**;** // max value to be a NN

**if** **((**lowerWord**.**EndsWith**(**"\'s"**)** **||** lowerWord**.**EndsWith**(**"s\'"**)** **||**

lowerWord**.**EndsWith**(**"s"**))** **&&** currentTag **==** "NN"**)**

occurenceAdder **+=** **(**double**)**bestValueWeight**;**

**if** **(**lowerWord**.**Contains**(**"."**)** **&&** currentTag **==** "NN"**)**

occurenceAdder **+=** **(**double**)**worstValueWeight **/** 2**;**

**if** **((**lowerWord**.**Contains**(**"-"**)** **||** lowerWord**.**Contains**(**"/"**))** **&&**

currentTag **==** "NN"**)**

occurenceAdder **+=** **(**double**)**worstValueWeight **/** 2**;**// NN

**if** **((**lowerWord**.**Contains**(**"-"**)** **||** lowerWord**.**Contains**(**"/"**))** **&&**

currentTag **==** "JJ"**)**

occurenceAdder **+=** **(**double**)**worstValueWeight **/** 2**;** // JJ

**if** **((**lowerWord**.**Contains**(**"-"**)** **&&** lowerWord**.**Count**(**x **=>** x **==** '-'**)** **>** 2**)** **&&**

currentTag **==** "OT"**)**

occurenceAdder **+=** **(**double**)**worstValueWeight **/** 2**;**

// OT (e.g.: At-the-central-library)

**if** **(**lowerWord**.**Contains**(**"/"**)** **&&** currentTag **==** "OT"**)**

occurenceAdder **+=** **(**double**)**worstValueWeight **/** 2**;** // OT

**if** **(**lowerWord**.**EndsWith**(**"\'t"**)** **&&** currentTag **==** "VB"**)**

occurenceAdder **+=** **(**double**)**bestValueWeight**;**

**if** **((**lowerWord**.**EndsWith**(**"\'ve"**)** **||** lowerWord**.**EndsWith**(**"\'ll"**))** **&&**

currentTag **==** "PN"**)**

occurenceAdder **+=** **(**double**)**bestValueWeight**;**

Parametrul currentTag este cel care se verifica la momentul intrarii in functia de recunoastere a tag-ului pentru cuvantul (testWord) necunoscut, functia de decodificare va incerca sa testeze fiecare tag inafara de cel de sfarsit/inceput de propozitie, si cel cu cea mai mare probabilitate va fi ales.

**Unknown word final function**

Pentru a putea combina aceste componenente va trebuii sa calculam probabilitatea cuvantului necunoscut cu tagul curent in functie de sufixele si prefixele cu care acesta este compus si probabilitatea in functie de conditiile adunate in ponderea de reguli. Acestea sunt combinate intr-o probabilitate finala:

|  |  |
| --- | --- |
|  | (3.14) |

Unde, – probabilitatea cuvantului necunoscut sa fie asociat cu tag-ul

– formula de calcul a sufixelor si a prefixelor cuvantului necunoscut , asociate cu tag-ul

– formula de calcul pe baza regulilor trecute pentru cuvantul necunoscut , asociat cu tag-ul

Formula se calculeaza prin a aduna probabilitatea sufix-ului si a prefix-ului, daca exista prefix/sufix pentru cuvantul si tag-ul curent , mai apoi fiind normalizata cu formula min-max de la (3.15) in intervalul [0, 2]. Daca nu gaseste sufix sau prefix, atunci va cauta cea mai mica valoare pentru sufix & prefix si va face aceelasi proces dar dupa normalizare, va inmulti rezultatul si cu o constanta egala cu 0.01.

Formula se calculeaza prin a aduna ponderile pentru regulile care le trece (occurenceAdder aduna aceste ponderi, variabila intalnita anterior), este normalizata tot cu formula de la (3.15) dar cu limita superioara egala cu variabila bestValueWeight, adica intervalul [0, 2.5]. Pentru cazul cand nu gaseste nicio regula si adunarea ponderilor este 0, atunci aceasta normalizeaza in intervalul [0, 2.5] valorea de la variabila worstValueWeight adica 1.5, mai apoi rezultatul fiind si el inmultit cu aceeasi constanta egala cu 0.01.

La final cand se vor aduna aceste componente rezultatul poate depasi intervalul probabilitatii, de aceea se executa o functie care converteste valoare la maximul limitei superioare adica 1.0. Acest lucru nu este gresit deoarece, daca trece de limita maxima de 1.0 atunci inseamna ca e o probabilitate de 100% sa fie tag-ul respectiv (deobicei aceasta valoare este trecuta pentru tag-ul de substantiv, care de cele mai multe ori este tag-ul corect). Formula aceasta (3.14) este foarte importanta deoarece ea combina cele 2 formule astfel incat sa poata da o probabilitate chiar daca nu gaseste sufixe/prefixe sau reguli (probabilitate foarte mica), dar sa poata da si o probabilitate buna daca nu gaseste o sufixe/prefixe dar gaseste reguli sau invers daca gaseste sufixe/prefixe dar nu gaseste reguli.

Formula normalizarii min-max:

|  |  |
| --- | --- |
|  | (3.15) |

Unde, noua valoare

vechea valoare

min – limita inferioara

max – limita superioara

ex. 1. cuvantul necunoscut = romana si tag-ul = “NN”

nu are niciun sufix sau prefix recunoscut in engleza deci presupunem ca suma valorii probabilitatii minime a sufix-ului si a prefix-ului este = 0.17 (0.10 = sufix, 0.7 = prefix)

Folosind formula de la (3.15) avem:

|  |  |
| --- | --- |
|  | (3.16)  (3.17) |

Cuvantul “romana” nu are nicio regula recunoscuta in engleza, aplicand tot (3.15):

|  |  |
| --- | --- |
| Aplicand formula de la (3.14) obtinem: | (3.18)  (3.19)  (3.20) |

Se poate vedea cum probabilitatea este foarte mica daca nu se gaseste nicio regula si niciun sufix/prefix.

ex. 2. Cuvantul necunoscut = Romanian si tag-ul = “NN”

In acest caz, sufixul cuvantului este “ian”, luand probabilitatea din model-ul aplicatiei, avem suma = 0.239 (0.239 = sufix, 0.0 = prefix):

|  |  |
| --- | --- |
|  | (3.21) |

Aici nu mai inmultim cu 0.01 deoarece am gasit un sufix in setul antrenat.

La reguli, se identifica una, cuvantul incepe cu litera mare. Deci valoarea de adunare a ponderilor:

|  |  |
| --- | --- |
|  | (3.22)  (3.23)  (3.24) |

Cuvantul Romanian are o probabilitate foarte mare sa fie asociat cu tag-ul de substantiv. Cu toate ca din acest exemplu reiese ca Romanian este substantiv (pentru cazul “The Romanian who won the nobel prize is ..” atunci functia returneaza rezultatul corect) dar in majoritatea cazurilor el este adjectiv (“My Romanian friends ..”) . Ca functia sa dea cel mai bun rezultat, ea trebuie combinata cu o probabilitate de tranzitie, acest lucru se va prezenta in sub-capitolul urmator numit “Decoder”.

**Implementare Unknown words function**

Mai intai se vor obtine probabilitatile sufixelor si a prefixelor pentru cuvantul necunoscut si tag-ul curent si dupa aceea se va calcula suma bazata pe ponderile regulilor. Primul pas este sa obtinem minima valoare pentru sufix si prefix, dupa care putem cauta prefixul si sufixul cuvantului. Vom incepe cu sufixele si prefixele cuvintelor care incep cu litera mare iar daca nu le gasim acolo, le cautam in lista cu sufixe si prefixe cu litera mica. Cautare minim:

// founding capitalized prefix min value

**foreach** **(**var pfx **in** **this.**PrefixCapitalizedWordEmissionProbabilities**)**

**{**

**foreach** **(**var pf **in** pfx**.**TagFreq**)**

**{**

**if** **(**pf**.**Value **<** minPrefix**)**

minPrefix **=** pf**.**Value**;**

**}**

**}**

// founding capitalized suffix min value

**foreach** **(**var sfx **in** **this.**SuffixCapitalizedWordEmissionProbabilities**)**

**{**

**foreach** **(**var sf **in** sfx**.**TagFreq**)**

**{**

**if** **(**sf**.**Value **<** minSuffix**)**

minSuffix **=** sf**.**Value**;**

**}**

**}**

Cautare sufix & prefix pentru cuvantul necunoscut de test:

**foreach** **(**var pfx **in** **this.**PrefixCapitalizedWordEmissionProbabilities**)**

**{**

**if** **(**lowerWord**.**StartsWith**(**pfx**.**Word**))**

**{**

**if** **(**pfx**.**TagFreq**.**ContainsKey**(**currentTag**))**

**{**

preffixVal **=** pfx**.**TagFreq**[**currentTag**];**

**break;**

**}**

**}**

**}**

**foreach** **(**var sfx **in** **this.**SuffixCapitalizedWordEmissionProbabilities**)**

**{**

**if** **(**lowerWord**.**EndsWith**(**sfx**.**Word**))**

**{**

**if** **(**sfx**.**TagFreq**.**ContainsKey**(**currentTag**))**

**{**

suffixVal **=** sfx**.**TagFreq**[**currentTag**];**

**break;**

**}**

**}**

**}**

Dupa cum se poate vedea, odata gasit sufixul/prefixul, vom iesi din iteratie deoarece nu trebuie sa mai cautam. Metoda .ContainsKey(..) verifica daca exista cheia cu tag-ul curent in sufixul/prefixul respectiv fara a mai trebuii sa iteram si dictionarul de tag-uri. Algoritmul pentru sufixele/prefixele cu litera mica este identic.

Dupa ce am terminat sa obtinem valorile, putem trece mai departe la calculul sumei si la normalizare:

double sum **=** **(**double**)**preffixVal **+** suffixVal**;**

double minSum **=** **(**double**)(**minPrefix **+** minSuffix**);**

const double higherWordFixBound **=** 2.0d**;**

**if** **(**sum **==** 0.0d**)**

**{**

double minProbabilityForZero **=** TextPreprocessing**.**Normalization**.**

MinMaxNormalization**(**minSum**,** 0.0d**,** higherWordFixBound**) \*** zeroProbabilityDifferenceToMinProbability**;** // 2.0d

proc **+=** minProbabilityForZero**;**

**}**

**else**

**{**

proc **+=** **(**double**)**TextPreprocessing**.**Normalization**.**

MinMaxNormalization**(**sum**,** 0.0d**,** higherWordFixBound**);** // 2.0d

**}**

Variabila proc este initiata cu 0 si este calculeaza formula de la (3.14), iar constanta zeroProbabilityDifferenceToMinProbability este 0.01. Functia min-max normalization este implementata in sub-clasa Normalization a clasei principale TextPreprocessing astfel:

**public** static double MinMaxNormalization**(**double x**,** double min**,** double max**)**

**{**

**return** **(**double**)(**x **-** min**)** **/** **(**max **-** min**);**

**}**

Pentru formula bazata pe reguli, regulile au fost scrise in sectiunea anterioara “Rule-based tag weights”, formula de calcul a ponderilor este implementata astfel:

**if** **(**occurenceAdder **==** 0.0d**)**

**{**

double minProbabilityForZero **=** TextPreprocessing**.**Normalization**.**

MinMaxNormalization**(**lowerAdderBound**,** 0**,** higherAdderBound**)** **\*** zeroProbabilityDifferenceToMinProbability**;**

proc **+=** minProbabilityForZero**;**

**}**

**else**

proc **+=** TextPreprocessing**.**Normalization**.**

MinMaxNormalization**(**occurenceAdder**,** 0**,** higherAdderBound**);**

Unde lowerAdderBound si lowerAdderBound sunt egale cu bestValueWeight (2.5) respectiv worstValueWeight (1.5). Inainte de a returna rezultatul, trebuie sa convertim valoarea din proc la o probabilitate, deci:

proc **=** TextPreprocessing**.**Normalization**.**BoundProbability**(**proc**);**

unde BoundProbability este implementata tot in clasa Normalization astfel:

**public** static double BoundProbability**(**double x**)**

**{**

**if** **(**x **>** 1.0d**)**

**return** 1.0d**;**

**else** **if** **(**x **<** 0.0d**)**

**return** 0.0d**;**

**else** **return** x**;**

**}**

Dupa aceasta verificare si conversie, putem returna rezultatul probabilitatii cuvantului necunoscut de intrare, asociat tag-ului curent de verificare.

### Decoder

In sub-capitolul anterior am prezentat formarea modelului pentru cuvinte cunoscute si cuvinte necunoscute, probabilitati de emisie si de tranzitie. In acest sub-capitol, se prezinta partea de decodor a modelului, fara acesta nu s-ar putea putea determina secventa variabilelor ascunse (secventa tag-urilor) asociate cu secventa de observatii (cuvintele unei propozitii) [1].

Un algoritm important de decodificare a unui model ascuns, pe baza programarii dinamice, este algoritmul lui Viterbi. Algoritmul lui Viterbi poate procesa starile trellis-ului pornind de la stanga la dreapta dar de asemenea poate sa o faca si invers. Am sa numesc aceste metode forward (merge inainte de la primul cuvant din propozitie pana la sfarsit de propozitie) si backward (merge de la sfarsitul propozitiei la inceputul acesteia) si bidirectional (o combinatie intre ambele), acestea formeaza algoritmul de decodificare Viterbi. Formula generala de calculare a fiecarui nod este:

|  |  |
| --- | --- |
|  | (3.16) |

unde, probabilitatea nodului curent

probabilitatea nodului anterior la pasul de timp anterior

– probabilitatea de tranzitie de la starea (tag-ul) anterioara la starea curenta

– probabilitatea de emisie (sau state observation likelihood) a observatiei simbolului

(token/cuvant/caracter) dandu-se starea j curenta

Formula prezentata la (3.16) calculeaza probabilitatea maxima de trecere de la o parte de vorbire la alta, aceasta calculeaza probabilitatea si cu nodul anterior asta daca exista o legatura bigram/trigram intre tag-ul nodului anterior si tag-ul nodului curent, altfel nu necesita calcularea fiecarui nod de tag deoarece rezultatul ar fi 0.

Un nod Viterbi este implementat astfel in program:

**public** class ViterbiNode

**{**

**public** double value**;**

**public** string CurrentTag**;**

**public** ViterbiNode PrevNode**;**

**public** ViterbiNode NextNode**;** // + bidirectionality

**public** ViterbiNode**(**double value**,** string CurrentTag**,**

ViterbiNode PrevNode **=** **null,** ViterbiNode NextNode **=** **null)**

**{**

**this.**value **=** value**;**

**this.**CurrentTag **=** CurrentTag**;**

**this.**PrevNode **=** PrevNode**;**

**this.**NextNode **=** NextNode**;**

**}**

**}**

Aceasta se foloseste de obiectul PrevNode pentru a putea face backtack la nodul anterior, astfel pentru a putea decodifica intreaga secventa. NextNode se foloseste pentru metoda de backward, iar ambele sunt folosite pentru metoda in ambele directii. Metodele nu calculeaza formula si pentru noduri unde rezultatul va fi zero (nu exista probabilitate anterioara/emisie/tranzitie).

Functia care implementeaza algoritmul lui Viterbi este implementata astfel:

**public** void ViterbiDecoding**(**PartOfSpeechModel tagger**,**

List**<**Tokenizer**.**WordTag**>** testWords**,**

string modelForward **=** "bigram"**,**

string modelBackward **=** "bigram"**,**

string mode **=** "forward"**)**

**{**

**this.**UnknownWords **=** **new** HashSet**<**string**>();**

**this.**ForwardHistory **=** **new** List**<**ViterbiNode**>();**

**this.**BackwardHistory **=** **new** List**<**ViterbiNode**>();**

**this.**PredictedTags **=** **new** List**<**string**>();**

**this.**ViterbiGraph **=** **new** List**<**List**<**ViterbiNode**>>();**

**if** **(**mode**.**Equals**(**"forward"**)** **||** mode**.**Equals**(**"f+b"**))**

**this.**ForwardAlgorithm**(**tagger**,** testWords**,** modelForward**);**

**if** **(**mode**.**Equals**(**"backward"**)** **||** mode**.**Equals**(**"f+b"**))**

**this.**BackwardAlgorithm**(**tagger**,** testWords**,** modelBackward**,** mode**);**

**if** **(**mode**.**Equals**(**"f+b"**))**

**this.**BiDirectionalModelTrace**();**

TextPreprocessing**.**Cleaning**.**EliminateAllEndOfSentenceTags**(ref** testWords**);**

**}**

Dupa cum se poate vedea, functia primeste 5 parametrii de intrare, primul primeste informatii despre model, al doilea sunt colectia setului de testare, al treilea si al patrulea specifica ce tip de tranzitie se poate selecta (bigram/trigram) si ultimul specifica metoda de decodare (forward, backward, bidirectional). Codul selectat implementeaza metoda care sterge token-urile (cuvintele) care au tag de final/inceput de propozitie.

#### 3.2.4.1 Forward method

In aceasta metoda, propozitiile sunt procesate de la stanga la dreapta. Cand se ajunge la nodul final cu valoarea cea mai mare, se face un backtrace pentru a returna etichetele finale. Metoda forward foloseste formula de la (3.16). Metoda Forward poate sa foloseasca probabilitatea de tranzitie bigram dar atat si trigram, totusi, pentru trigram aceasta nu va putea sa calculeze nodul de inceput, de aceea, aceasta foloseste probabilitatea de la bigram pentru a calcula probabilitatea primului nod. Implementarea acestei metode si a celei backward este lunga si complicata, de aceea se vor prezenta implementarea acestora pe bucati.

Metoda forward va prin a initia flag de start de propozitie noua, un iterator pentru a recunoaste daca putem aplica trigram (in cazu in care acesta este activat din functie) si va itera de la 0 la N, unde N este ultimul token din setul de testare. La fiecare iteratie se va verifica flag-ul de start, va creste iteratorul pentru trigram si vom obtine probabilitatea de emisie pentru token-ul (cuvantul) care il modificam la timpul i.

bool startPoint **=** **true;**

int triPoz **=** **-**1**;**

**for** **(**int i **=** 0**;** i **<** testWords**.**Count**;** i**++)** // starting from left (0 index)

**{**

triPoz**++;**

**if** **(**testWords**[**i**].**tag **==** "."**)** // we can verify word instead of tag here

**{**

Backtrace**(**method**:** "forward"**);** // decompress method, going from right to left using prev nodes, applied only when '.' is met

startPoint **=** **true;**

**continue;**

**}**

var foundWord **=** tagger**.**WordCapitalizedTagsEmissionProbabilities**.**

Find**(**x **=>** x**.**Word **==** testWords**[**i**].**word**);**

**if** **(**foundWord **==** **null)**

foundWord **=** tagger**.**WordTagsEmissionProbabilities**.**

Find**(**x **=>** x**.**Word **==** testWords**[**i**].**word**.**ToLower**());**

**......**

In cazul in care startPoint este true, atunci se cunoaste faptul ca verificam defapt tranzitia de la inceput de propozitie (tag-ul de ‘.’) la un alt tag care nu este inceput/sfarsit de propozitie si nu putem aplica probabilitatea de tranzitie a trigramului (ea poate fi aplicata in cazul cand triPoz >= 2). Aceasta mai intai verifica daca exista o tranzitie bigram de trecere de la tag-ul de inceput de propozitie la tag-ul cuvantului gasit la timpul i (foundWord), daca gaseste atunci va itera prin tag-urile acestuia, daca nu va gasi, atunci va itera prin toate probabilitatile de tranzitie bigram de la tag-ul de inceput de propozitie la alt tag diferit.

**if** **(**startPoint**)** // first node (start)

**{**

triPoz **=** 0**;**

List **<**ViterbiNode**>** vList **=** **new** List**<**ViterbiNode**>();**

**if(**foundWord **!=** **null)**

**if** **(**foundWord**.**TagFreq**.**Count **==** 1 **&&** foundWord**.**TagFreq**.**ContainsKey**(**"."**))**

foundWord **=** **null;**

**if** **(**foundWord **==** **null)**

**{**

UnknownWords**.**Add**(**testWords**[**i**].**word**);**

// we take the best transition case where first item is "."

// case 2: all the transitions

var orderedTransitions **=** tagger**.**BigramTransitionProbabilities**.**

OrderByDescending**(**x **=>** x**.**Value**).**ToList**();**

double product **=** 0.0d**;**

string nodeTag **=** "NULL"**;**

**foreach** **(**var item **in** orderedTransitions**)**

**if** **(**item**.**Key**.**Item1**.**Equals**(**"."**)** **&&** item**.**Key**.**Item2 **!=** "."**)**

**{**

double uniVal **=** tagger**.**UnigramProbabilities**.**FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**item**.**Key**.**Item2**)).**Value**;**

double biTrans **=** **(**double**)(**uniVal **\*** tagger**.**BgramLambda1**)** **+**

**(**item**.**Value **\*** tagger**.**BgramLambda2**);**

double unknownProcent **=** tagger**.**

**G**etValueWeightForUnknownWord**(**testWords**[**i**].**word**,** item**.**Key**.**Item2**);**

product **=** biTrans **\*** unknownProcent**;**

nodeTag **=** item**.**Key**.**Item2**;**

ViterbiNode node **=** **new** ViterbiNode**(**product**,** nodeTag**);**

vList**.**Add**(**node**);**

**}**

**}**

**.....**

Atunci cand nu gaseste cuvantul in lista cu probabilitati de emisie, se va adauga cuvantul necunoscut in lista de cuvinte necunoscute (ne va ajuta mai tarziu pentru metrici de acuratete), se va itera lista cu probabilitati de tranzitie bigram si se va cauta ca ca al doilea tag sa fie diferit de tag-ul de inceput/sfarsit de propozitie. In momentul ce gaseste unul, va calcula variabila biTrans ca adunare intre probabilitatea de bigram (inmultita la randul ei cu ponderea lambda de bigram, calculata si explicata anterior la interpolarea liniara) si probabilitatea de unigram (la fel inmultita cu ponderea lambda de unigram). Dupa aceasta, se va calcula proabilitatea cuvantului necunoscut (descrisa la capitolul anterior), cu tag-ul curent si se va salva in variabila unknownProcent, valoarea finala este, in sfarsit, calculata ca produs din probabilitatea de tranzitie si probabilitatea cuvantului necunoscut.

Dupa ce aceasta este calculata, se poate salva nodul cu valoarea probabilitatii pentru tag-ul curent, si tag-ul curent in lista cu noduri pentru iteratia la timpul i. Nodurile la timpul i-1 si i+1 sunt null la inceput. Daca totusi se gaseste cuvantul/token-ul in lista cu probabilitati de emisie atunci se va executa urmatoarea conditie:

**else**

**{**

**foreach** **(**var wt **in** foundWord**.**TagFreq**)**

**{**

**if** **(**wt**.**Key **==** "."**)**

**continue;**

double emissionFreqValue **=** wt**.**Value**;** // eg. Jane -> 0.1111 (NN)

Tuple**<**string**,** string**>** tuple **=** **new** Tuple**<**string**,** string**>(**"."**,** wt**.**Key**);**

Double biTransition **=** tagger**.**BigramTransitionProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**tuple**)).**Value**;** // eg. NN->VB - 0.25

Double uniVal **=** tagger**.**UnigramProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**wt**.**Key**)).**Value**;**

double biTrans **=** **(**double**)(**uniVal **\*** tagger**.**BgramLambda1**)** **+**

**(**biTransition **\*** tagger**.**BgramLambda2**);**

double product **=** **(**double**)**emissionFreqValue **\*** biTrans**;**

ViterbiNode node **=** **new** ViterbiNode**(**product**,** wt**.**Key**);**

vList**.**Add**(**node**);**

**}**

**}**

**this.**ViterbiGraph**.**Add**(**vList**);**

startPoint **=** **false;**

Aceasta secventa va executa asemanator cu secventa anterioara cand nu gaseste cuvantul/token-ul in lista de emisii, doar ca aici se vor itera doar tag-urile token-ului actual ci nu toate. Se poate observa ca la produs aici nu mai intervine probabilitatea cuvantului necunoscut ci doar probabilitatea de emisie pentru fiecare tag de emisie al cuvantului curent, inmultit cu probabilitatea de tranzitie descrisa anterior.

In continuare, dupa ce calculam lista de noduri la timpul i, o vom adauga in matricea 2d dinamica (lista de lista) ViterbiGraph si vom seta startPoint pe false deoarece acum am trecut de primul cuvant din propozitie si putem sa trecem la conditia unde avem si un nod anterior si putem aplica si trigram daca functia a fost apelata cu acest parametru. Pentru conditia cand nu mai suntem in start, putem la fel sa gasim cuvantul in lista cu probabilitati de emisie sau sa nu il gasim, atunci cand nu gasim il gasim si aplicam trigram, vom avea:

**if** **(**foundWord **==** **null)**

**{**

UnknownWords**.**Add**(**testWords**[**i**].**word**);**

**for** **(**int j **=** 0**;**

j **<** **this.**ViterbiGraph**[this.**ViterbiGraph**.**Count **-** 1**].**Count**;** j**++)**

**{**

ViterbiNode vGoodNode **=** **new** ViterbiNode**(**0.0d**,** "NULL"**);**

ViterbiNode elem **=** **this.**ViterbiGraph**[this.**ViterbiGraph**.**Count **-** 1**][**j**];**

// we take the best transition case where first item is "."

var orderedTransitions **=** tagger**.**BigramTransitionProbabilities**.**

OrderByDescending**(**x **=>** x**.**Value**).**ToList**();**

**if** **(**model **==** "trigram" **&&** triPoz **>=** 2**)**

**{**

**if** **(**elem**.**PrevNode **==** **null)**

**continue;**

ViterbiNode elem2 **=** elem**.**PrevNode**;**

var orderedTransitionsTri **=** tagger**.**TrigramTransitionProbabilities**.**

OrderByDescending**(**x **=>** x**.**Value**).**ToList**();**

double product **=** 0.0d**;**

string nodeTag **=** "NULL\_TRI"**;**

**foreach** **(**var item **in** orderedTransitionsTri**)**

**if** **(**item**.**Key**.**Item1**.**Equals**(**elem2**.**CurrentTag**)** **&&**

item**.**Key**.**Item2**.**Equals**(**elem**.**CurrentTag**)** **&&**

item**.**Key**.**Item3 **!=** "."**)**

**{**

Tuple**<**string**,** string**>** biTuple **=** **new** Tuple**<**string**,** string**>**

**(**elem**.**CurrentTag**,** item**.**Key**.**Item3**);**

double biVal **=** tagger**.**BigramTransitionProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**biTuple**)).**Value**;**

double uniVal **=** tagger**.**UnigramProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**item**.**Key**.**Item3**)).**Value**;**

double triTransition **=** **(**double**)(**tagger**.**TgramLambda3 **\*** item**.**Value**)** **+**

**(**tagger**.**TgramLambda2 **\*** biVal**)** **+**

**(**tagger**.**TgramLambda1 **\*** uniVal**);**

double unknownProcent **=** tagger**.**GetValueWeightForUnknownWord**(**

testWords**[**i**].**word**,**

item**.**Key**.**Item3**);**

product **=** **(**double**)**elem**.**value **\*** triTransition **\*** unknownProcent**;**

nodeTag **=** item**.**Key**.**Item3**;**

**if** **(**product **>=** vGoodNode**.**value**)**

**{**

vGoodNode **=** **new** ViterbiNode**(**product**,** nodeTag**,** PrevNode**:** elem**);**

**}**

**}**

**}**

In acest caz, mai intai vom adauga cuvantul necunoscut in lista, si vom itera toate nodurile anterioare create, pentru a putea folosi formula descrisa la (3.16). Chiar daca vom itera toate nodurile, doar un singur nod va face legatura cu nodul curent, acela fiind nodul cu cea mai mare probabilitate de transfer. Daca pozitia triPoz este 2 (adica putem aplica trigram deoarece am trecut de sectiunea unde nu exista trigram) si modelul a fost ales ca trigram, atunci conditia se va indeplini si se va calcula probabilitatea pe nodul respectiv cu probabilitate de tranzitie de tip trigram. vGoodNode va fi nodul care va avea informatii despre nodul maxim calculat, procesul de calculare fiind asemanator cu cel descris anterior doar ca aici vom calcula variabila triTransition cu fiecare tranzitie (unigram, bigram, trigram) cu ponderile lambda de interpolare a fiecaruia. Dupa ce s-a calculat probabilitatea de tranzitie interpolata, se va calcula, asemanator anterior, probabilitatea cuvantului necunoscut pentru tag-ul curent.

Odata ce am calculat acestea, putem calcula probabilitatea nodului din formula (3.16), inmultind la cele calculate anterior si probabilitatea nodului anterior (adica nodul salvat in matricea dinamica 2d la linia ‘this.ViterbiGraph.Count – 1’ si coloana curenta ‘j’. Daca acest produs este mai mare ca vGoodNode, atunci vom inlocuii valoarea veche a acestui nod cu valoarea calculata curent si vom salva si obiectul elem, acesta fiind nodul curent care la timpul ‘i+1’ va deveni nodul precedent. Se poate observa in conditia subliniata se vor lua doar tranzitiile unde primul element din tuplul trigram este tag-ul obiectului cu contextul anterior nodului curent, tag-ul nodului curent si orice tag care nu marcheaza final de propozitie, aceasta facand legatura doar cu nodul curent si nu cu toate nodurile existente si posibile nodului anterior nodului curent. De asemenea, pentru modul forward, se poate observa ca elementele trigramului sunt luate de la stanga la dreapta, asta semnificand metoda inainte adica de la inceputul propozitiei spre finalul ei.

Pentru conditia de bigram, logica desfasurarii este foarte asemanatoare cu cea descrisa anterior:

double product **=** 0.0d**;**

string nodeTag **=** "NULL\_BI"**;**

**foreach** **(**var item **in** orderedTransitions**)**

**if** **(**item**.**Key**.**Item1**.**Equals**(**elem**.**CurrentTag**)** **&&** item**.**Key**.**Item2 **!=** "."**)**

**{**

double uniVal **=** tagger**.**UnigramProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**item**.**Key**.**Item2**)).**Value**;**

double biTrans **=** **(**double**)(**uniVal **\*** tagger**.**BgramLambda1**)** **+**

**(**item**.**Value **\*** tagger**.**BgramLambda2**);**

double unknownProcent **=** tagger**.**GetValueWeightForUnknownWord**(**

testWords**[**i**].**word**,** item**.**Key**.**Item2**);**

product **=** **(**double**)**elem**.**value **\*** biTrans **\*** unknownProcent**;**

nodeTag **=** item**.**Key**.**Item2**;**

**if** **(**product **>=** vGoodNode**.**value**)**

**{**

vGoodNode **=** **new** ViterbiNode**(**product**,** nodeTag**,** PrevNode**:** elem**);**

**}**

**}**

La final, dupa ce am calculat nodul cu valoarea maxima, il putem adauga in lista de noduri pentru iteratia la timpul ‘i’. Acest nod cu valoarea maxima se va calcula pentru fiecare nod calculat anterior.

Ultimul caz care a mai ramas, este atunci cand nu se verifica primul cuvant si cuvantul/token-ul curent este in lista de probabilitati de emisie, acest caz este asemanator cu cel descris anterior pentru cuvinte necunoscute, doar ca in loc de calcularea probabilitatii cuvantului necunoscut cu fiecare tag, se va calcula/prelua probabilitatea de emisie pentru tag-urile cuvantului. Implementarea pentru trigram este urmatoarea:

**foreach** **(**var tf **in** foundWord**.**TagFreq**) {**

**......**

**......**

**foreach** **(**ViterbiNode vn **in** **this.**ViterbiGraph**[this.**ViterbiGraph**.**Count **-** 1**])**

**{**

**if(**model **==** "trigram" **&&** triPoz **>=** 2**)**

**{**

**if** **(**vn**.**PrevNode **==** **null)**

**continue;**

Tuple**<**string**,**string**,**string**>** triTuple **=** **new** Tuple**<**string**,**string**,**string**>**

**(**vn**.**PrevNode**.**CurrentTag**,** vn**.**CurrentTag**,** tf**.**Key**);**

double triVal **=** tagger**.**TrigramTransitionProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**triTuple**)).**Value**;**

Tuple**<**string**,** string**>** biTuple **=** **new** Tuple**<**string**,** string**>**

**(**vn**.**CurrentTag**,** tf**.**Key**);**

double biVal **=** tagger**.**BigramTransitionProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**biTuple**)).**Value**;**

double uniVal **=** tagger**.**UnigramProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**tf**.**Key**)).**Value**;**

double triTransition **=** **(**double**)(**tagger**.**TgramLambda3 **\*** triVal**)** **+**

**(**tagger**.**TgramLambda2 **\*** biVal**)** **+**

**(**tagger**.**TgramLambda1 **\*** uniVal**);**

double product **=** **(**double**)**vn**.**value **\*** triTransition **\*** tf**.**Value**;**

**if(**product **>=** vGoodNode**.**value**)**

**{**

vGoodNode **=** **new** ViterbiNode**(**product**,** tf**.**Key**,** PrevNode**:** vn**);**

**}**

**}**

**}**

**}**

Valoarea tf.Value este probabilitatea de emisie pentru tag-ul curent al cuvantului actual, vn.value fiind probabilitatea nodului anterior. Implementarea bigram se face in conditia cand modelul nu este trigram:

Tuple**<**string**,** string**>** tuple **=** **new** Tuple**<**string**,** string**>**

**(**vn**.**CurrentTag**,** tf**.**Key**);**

double biTransition **=** tagger**.**BigramTransitionProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**tuple**)).**Value**;** // eg. NN->VB - 0.25

double uniVal **=** tagger**.**UnigramProbabilities**.**

FirstOrDefault**(**x **=>** x**.**Key**.**Equals**(**tf**.**Key**)).**Value**;**

double biTrans **=** **(**double**)(**uniVal **\*** tagger**.**BgramLambda1**)** **+**

**(**biTransition **\*** tagger**.**BgramLambda2**);**

double product **=** **(**double**)**vn**.**value **\*** biTrans **\*** tf**.**Value**;**

**if** **(**product **>=** vGoodNode**.**value**)**

**{**

vGoodNode **=** **new** ViterbiNode**(**product**,** tf**.**Key**,** PrevNode**:** vn**);**

**}**

vGoodNode este, la fel, introdus in lista de noduri actuale, la fel ca in procesul descris anterior, dupa aceasta lista este si ea introdusa la randul ei in matricea grafului Viterbi pentru timpul ‘i’, in final, ultima linie din matrice fiind sortata descrescator pentru ca primul nod sa fie cel cu probabilitatea cea mai mare atunci cand este realizat procesul de backtracing.

Functia de backtracing care se aplica si pentru metoda forward si backward este urmatoarea:

**private** void Backtrace**(**string method**)**

**{**

Var lastElement **=** **this.**ViterbiGraph**[this.**ViterbiGraph**.**Count **-** 1**][**0**];**

List**<**string**>** tagsViterbi **=** **new** List**<**string**>();**

**if(**method**.**Equals**(**"forward"**))**

**{**

ForwardHistory**.**Add**(**lastElement**);**

**while** **(true)**

**{**

**if** **(**lastElement**.**CurrentTag **!=** "."**)**

tagsViterbi**.**Insert**(**0**,** lastElement**.**CurrentTag**);**

**if** **(**lastElement**.**PrevNode **==** **null)**

**break;**

lastElement **=** lastElement**.**PrevNode**;**

**}**

**this.**PredictedTags**.**AddRange**(**tagsViterbi**);**

**}**

**else** **if(**method**.**Equals**(**"backward"**))**

**{**

BackwardHistory**.**Insert**(**0**,** lastElement**);**

**}**

**}**

Aceasta functie mai intai va selecta nodul cu valoarea cea mai mare (nodurile sunt sortate la fiecare iteratie de token), si va verifica care metoda este aleasa (forward/backward). Daca metoda a fost apelata cu parametru de forward atunci se va intra in conditia de forward, aici se va salva nodul final cu valoarea maxima intr-o lista de istorie, si se va face backtrack de la nodul curent pana cand prevNode este null. Toate aceste tag-uri provenite de la nodurile iterate se vor salva intr-o lista de tag-uri predictionate. Pentru metoda backward, metoda doar salveaza ultimul nod cu valoarea maxima in lista de istorie fara sa itereze nodurile precedente deoarece ordinea pentru metoda backward poate incurca ordinea in lista cu tag-uri predictionate.

#### 3.2.4.2 Backward method

Aceasta metoda presupune parcurgerea propozitiei/frazei de la dreapta spre stanga, adica de la sfarsit spre inceput. Aceasta metoda este de cele mai multe ori mult mai buna decat cea forward, ea fiind inversul procesului forward, o propozitie evaluata de la final spre inceput este mult mai bine evaluat decat de la inceput spre final. Totusi, propozitiile/frazele sunt de sine statatoare, ea poate itera fiecare propozitie din setul de antrenare in ordine (implementarea a fost facuta sa inceapa de la finalul setului de test spre inceput) dar fiecare propozitie va fi evaluata de la final spre inceput.

Backward method foloseste si ea probabilitatea de emisie, cuvintelor necunoscute si cea de tranzitie, doar ca la fel ca la forward, primul cuvant de la final nu va putea folosi modelul trigram. Inafara de asta, daca setul de testare este si el evaluat de la final spre inceput, atunci va trebuii sa verificam si momentul cand ajungem la primul cuvant/token in setul de testare dar sa sarim si peste ultimul token din setul de testare care are tag-ul asociat de sfarsit de propozitie (nu trebuie sa evaluam tag-ul de EOS), restul codului fiind aproape identic cu cel de la forward cu modificarile descrise aici. Inceputul metodei de backward descris in cod este urmatoarea:

**......**

**for** **(**int i **=** testWords**.**Count **-** 2**;** i **>=** **-**1**;** i**--)** // count - 2 is to start from the first word != "."

**{**

triPoz**++;**

**if** **(**i **==** **-**1**)** // we first check to see if we got to index -1

**{**

Backtrace**(**method**:** "backward"**);**

startPoint **=** **true;**

**continue;**

**}**

**if** **(**testWords**[**i**].**tag **==** "."**)**

**{**

Backtrace**(**method**:** "backward"**);**

startPoint **=** **true;**

**continue;**

**}**

**......**

Se poate observa ca iteratia incepe de la ultimul cuvant din setul de testare care nu are tag-ul de sfarsit de propozitie asociat, si merge pana la -1, adica pana la inceputul de propozitie pentru setul de testare. Indexul de -1 este verificat ca atunci stim ca putem face backtracing cu metoda backward si la fel si atunci cand dam de tag-ul de ’. ’, asta fiind la fel ca la metoda forward. La implementare, nodul anterior va fi acum cel posterior, iar probabilitatile de tranzitie vor fi evaluate de la final spre inceput precum:

**......**

**foreach** **(**var item **in** orderedTransitionsTri**)**

**if** **(**item**.**Key**.**Item3**.**Equals**(**elem2**.**CurrentTag**)** **&&**

item**.**Key**.**Item2**.**Equals**(**elem**.**CurrentTag**)** **&&**

item**.**Key**.**Item1 **!=** "."**)**

**......**

**if** **(**product **>=** vGoodNode**.**value**)**

**{**

vGoodNode **=** **new** ViterbiNode**(**product**,** nodeTag**,** NextNode**:** elem**);**

**}**

**......**

Codul descris aici este luat din conditia de trigram pentru cuvintele necunoscute, elementele tuplului de trigram sunt si ele evaluate de la cel din dreapta spre stanga. La final dupa ce s-a calculat produsul pentru nodul curent, se poate verifica daca e mai mare ca nodul maxim iar daca este atunci nodul maxim va deveni nodul curent cu legatura la nodul posterior (in acest caz fiind tot nodul anterior evaluat) la fel ca metoda de forward.

Cu toate ca aceasta metoda este foarte asemanatoare cu cea de forward, backtrace-ul pentru backward se face doar la final dupa ce s-a evaluat fiecare cuvant/token din setul de testare, daca s-ar face dupa fiecare propozitie ar fi mult mai complicat sa aranjam tag-urile predictionate in ordinea corecta.

In metoda de backtracking, se salveaza o lista cu nodurile finale cu valoarea maxima, iar cand setul de testare a fost iterat complet, atunci se pot emite tag-urile finale pentru metoda aceasta:

**if** **(**mode **==** "backward"**)**

**{**

**this.**PredictedTags **=** **new** List**<**string**>();**

List**<**ViterbiNode**>** historyCopy **=** **new** List**<**ViterbiNode**>(**BackwardHistory**);**

**for** **(**int i **=** 0**;** i **<** historyCopy**.**Count**;** i**++)**

**{**

List**<**string**>** tagsViterbi **=** **new** List**<**string**>();**

**while** **(true)**

**{**

**if** **(**historyCopy**[**i**].**CurrentTag **!=** "."**)**

tagsViterbi**.**Add**(**historyCopy**[**i**].**CurrentTag**);**

**if** **(**historyCopy**[**i**].**NextNode **==** **null)**

**break;**

historyCopy**[**i**]** **=** historyCopy**[**i**].**NextNode**;**

**}**

**this.**PredictedTags**.**AddRange**(**tagsViterbi**);**

**}**

**}**

Se poate observa ca aceasta conditie este si ea, la randul ei, foarte asemanatoare cu metoda de backtrace descrisa anterior la **Forward method**, diferenta aici fiind ca se va itera lista cu istoria nodurilor cu valoarea maxima iar backtrackingul se va face de la nodul curent spre nodul posterior (nextNode) pana cand acesta devine Null (adica am evaluat intreaga propozitie/fraza).

#### 3.2.4.3 Bidirectional method

Aceasta metoda combina ambele metode mentionate anterior, am descris in procesele anterioare ca se va pastra o lista de istoric cu nodurile finale pentru ambele metode, avand in vedere ca fiecare nod din lista de istoric reprezinta secventa de propozitie/fraza evaluata de metoda respectiva, putem atunci compara fiecare nod din cele 2 liste si doar atunci celui cu valoarea cea mai mare ii se va face backtrack pentru a obtine tag-urile finale predictionate.

Aceasta presupune ca valoarea cea mai mare a fost evaluata mai corect deoarece asta inseamna ca are o incredere mai mare acolo unde nodul este cu o probabilitate mai mare. Functia a metodei bidirectionale este urmatoarea:

**private** void BiDirectionalModelTrace**()**

**{**

**this.**PredictedTags **=** **new** List**<**string**>();**

**for(**int i **=** 0**;** i **<** BackwardHistory**.**Count**;** i**++)**

**{**

**if(**BackwardHistory**[**i**].**value **>** ForwardHistory**[**i**].**value**)**

**{**

List**<**string**>** tagsViterbi **=** **new** List**<**string**>();**

**while** **(true)**

**{**

**if** **(**BackwardHistory**[**i**].**CurrentTag **!=** "."**)**

tagsViterbi**.**Add**(**BackwardHistory**[**i**].**CurrentTag**);**

**if** **(**BackwardHistory**[**i**].**NextNode **==** **null)**

**break;**

BackwardHistory**[**i**]** **=** BackwardHistory**[**i**].**NextNode**;**

**}**

**this.**PredictedTags**.**AddRange**(**tagsViterbi**);**

**}**

**else**

**{**

List**<**string**>** tagsViterbi **=** **new** List**<**string**>();**

**while** **(true)**

**{**

**if(**ForwardHistory**[**i**].**CurrentTag **!=** "."**)**

tagsViterbi**.**Insert**(**0**,** ForwardHistory**[**i**].**CurrentTag**);**

**if** **(**ForwardHistory**[**i**].**PrevNode **==** **null)**

**break;**

ForwardHistory**[**i**]** **=** ForwardHistory**[**i**].**PrevNode**;**

**}**

**this.**PredictedTags**.**AddRange**(**tagsViterbi**);**

**}**

**}**

**}**

Pentru cazul cand valorile nodurilor sunt egale (foarte rar) atunci cel mai probabil se indica aceeleasi tag-uri pentru propozitia/fraza respectiva si oricare metoda s-ar alege, rezultatul evaluarilor ar fi la fel. Logica codului este la fel cu cea mentionata la backtracking pentru metoda forward si pentru metoda backward.

In urmatorul grafic, se poate vedea de cate ori se intra in branch-ul de backward, in cel de forward si de cate ori sunt valorile maxime pentru forward si backward sunt egale:

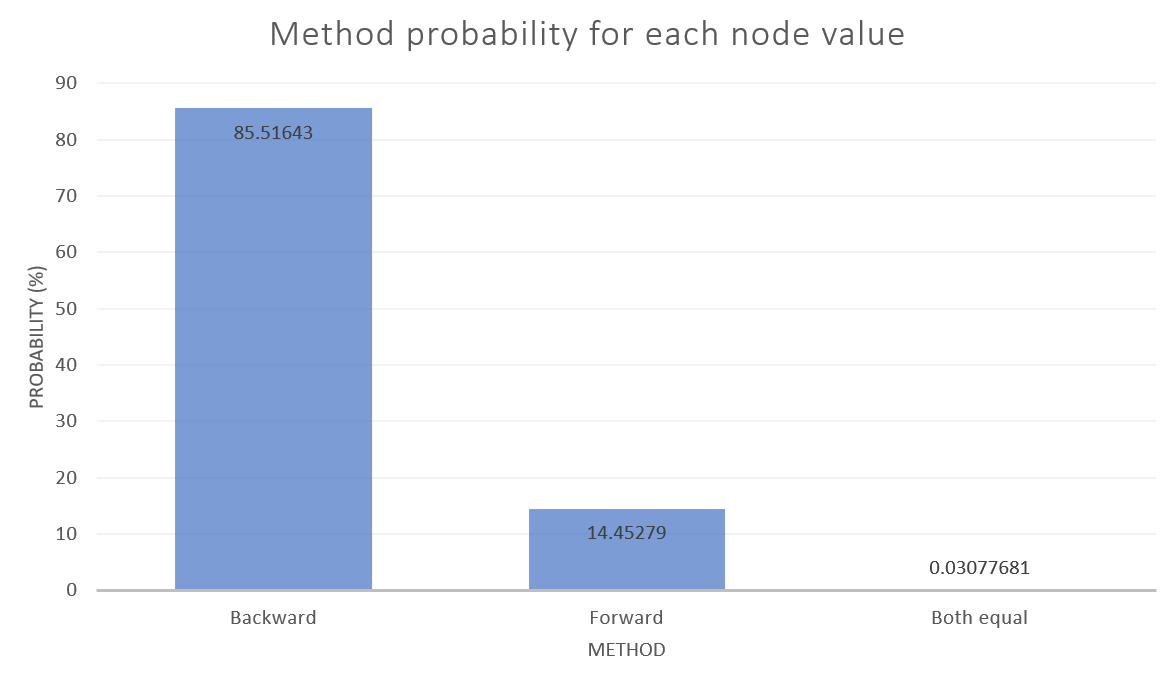


Figura 3.7 – probabilitatea fiecarei metode sa fie alese in functie de valoarea nodului final

### Evaluation

Dupa ce decodorul a emis tag-urile predictionate pentru setul de testare, va trebuii sa evaluam daca aceste tag-uri sunt corecte. Cum algoritmul acesta este unul de invatare supervizata, avem si informatii legate de tag-urile corecte a setului de testare, cu toate ca nu se folosesc aceste informatii, acestea sunt foarte folositoare atunci cand se doreste evaluarea modelului. Evaluarea pentru acest model este impartita in 2 categorii:

* Evaluarea acuratatii ”naive”
* Evaluarea matricii de confuzie

#### 3.2.5.1 Naive accuracy

Aceasta metoda simpla de evaluare presupune calcularea numarului de tag-uri predictionate corect pe (supra) numarul total de tag-uri predictionate. Aceasta este de fapt, acuratetea totala pentru toate cuvintele/token-urile din setul de testare. Pentru acest tip de model, acuratetea pentru cuvintele cunoscute si separat pentru cuvintele necunoscute este foarte utila deoarece putem vedea separat acuratetea modelului Markov si cea a modelului pentru cuvinte necunoscute.

Functia care implementeaza acuratetea naiva este urmatoarea:

**public** float GetNaiveAccuracy**(**List**<**Tokenizer**.**WordTag**>** testData**,**

List**<**string**>** predictedTags**,**

HashSet**<**string**>** unknownWords**,**

string evalMode **=** "k+u"**)**

**{**

int wordsHit **=** 0**;**

**int nrOfWords = 0;**

**for (int i = 0; i < testData.Count; i++)**

**{**

**if (evalMode != "k+u")**

**{**

**if (unknownWords.Contains(testData[i].word))**

**{**

**if (evalMode == "k")**

**continue;**

**}**

**else**

**{**

**if** **(**evalMode **==** "u"**)**

**continue;**

**}**

**}**

**if** **(**testData**[**i**].**tag **==** predictedTags**[**i**])**

wordsHit**++;**

nrOfWords**++;**

**}**

float accuracy **=** **(**float**)**wordsHit **/** nrOfWords**;**

**return** accuracy**;**

**}**

Aceasta functie are un parametru de intrare unde se poate seta ce fel de acuratete se doreste la returnare, ’k+u’ este acuratetea generala, ’k’ este acuratetea pentru cuvinte cunoscute si ’u’ este acuratetea pentru cuvinte necunoscute. Functia foloseste Set-ul de cuvinte necunoscute care a fost incarcat la decodare. Acuratetea finala se va calcula cu ca wordsHit (incrementat daca tag-ul la indexul i pentru setul de testare este aceelasi cu tag-ul predictionat la indexul i) supra numarul total de cuvinte. In momentul cand se alege acuratetea pentru cuvinte cunoscute, metoda verifica daca la indexul curent acest cuvant se afla si in setul de cuvinte necunoscute, daca nu se afla atunci acesta continua normal cu evaluarea wordsHit, daca se afla atunci se trece la indexul urmator fara sa fie procesat si acest cuvant/token. Pentru cuvintele necunoscute, procesul este asemanator pentru acuratetea cuvintelor necunoscute.

#### 3.2.5.2 Confusion matrix

Matricea de confuzie, cunoscuta si ca matricea de eroare, este un tabel specific care permite vizualizarea performantei unui algoritm de invatare supervizata. Liniile tabelului reprezinta clasa reala (tag-ul corect) iar coloanele reprezinta clasa predictionata (tag-ul predictionat de decodor) []. Acest tabel este foarte util atunci cand se evalueaza algoritmul pe mai multe clase (multi-class classification), obtinem metrici de evaluare pentru fiecare clasa iar metrica totala este media metricilor tuturor claselor.

****

Figura 3.8 – matricea de confuzie – informatii []

Fiecare clasa unica din setul de testare va avea propia ei matrice de confuzie, cele 4 valori tp, tn, fp, fn vor fi la inceput initializate cu 0 si vor creste in timp ce se itereaza setul de testare cu clasa predictionata.

Metricile de evaluare care se pot realiza in urma utilizarii unei matricii de confuzie si care sunt implementate in sistem sunt:

**Accuracy**: aceasta aici ia in calcul si cazurile cand clasa nu apare nici pentru setul de test si nici nu a fost predictionata (true negative)

|  |  |
| --- | --- |
|  | (3.17) |

**Precision**: este procentajul rezultatelor care sunt relevante. Aceasta incearca sa raspunda la intrebarea: “Ce proportie identificata pozitiv (tp) este si corecta?”

|  |  |
| --- | --- |
|  | (3.18) |

**Recall**: se refera la procentajul tuturor rezultatelor relevante corect clasificate de algoritm. Aceasta incearca sa raspunda la intrebarea: “Ce proportie identificata pozitiv (tp) a fost identificata corect?”

|  |  |
| --- | --- |
|  | (3.19) |

**Specificity (True Negative Rate)**: reprezinta proportia a rezultatelor negative, corect identificate ca negative.

|  |  |
| --- | --- |
|  | (3.20) |

**F-measure**: reprezinta media armonica dintre precizie si recall.

|  |  |
| --- | --- |
|  | (3.21) |

Deobicei, parametrul are valoarea 1, atunci noua formula se va numi **F1-Score**, ea fiind redusa la forma:

|  |  |
| --- | --- |
|  | (3.22) |

Implementarea acestor metrici de evaluare se face in functia urmatoare:

**public** void CreateSupervizedEvaluationsMatrix**(**

List**<**Tokenizer**.**WordTag**>** testData**,**

List**<**string**>** predictedTags**,**

HashSet**<**string**>** unknownWords**,**

string evalMode **=** "k+u"**,**

int fbeta **=** 1**)**

**{**

ClassTags **=** **new** HashSet**<**string**>();**

finalMatrix **=** **new** List**<**List**<**float**>>();**

**foreach** **(**var item **in** testData**)**

**this.**ClassTags**.**Add**(**item**.**tag**);**

**foreach** **(**string item **in** predictedTags**)**

**this.**ClassTags**.**Add**(**item**);**

**foreach(**var tag **in** **this.**ClassTags**)**

**{**

int tp **=** 0**,** fp **=** 0**,** fn **=** 0**,** tn **=** 0**;**

**for** **(**int i **=** 0**;** i **<** testData**.**Count**;** i**++)**

**{**

**if** **(**testData**[**i**].**tag **!=** tag **&&** predictedTags**[**i**]** **!=** tag**)**

tn**++;**

**else** **if** **(**testData**[**i**].**tag **==** tag **&&** predictedTags**[**i**]** **==** tag**)**

tp**++;**

**else** **if** **(**testData**[**i**].**tag **==** tag **&&** predictedTags**[**i**]** **!=** tag**)**

fn**++;**

**else** **if** **(**testData**[**i**].**tag **!=** tag **&&** predictedTags**[**i**]** **==** tag**)**

fp**++;**

**}**

float accuracy **=** **(**float**)(**tp **+** tn**)** **/** **(**tp **+** tn **+** fn **+** fp**);**

**if** **(**float**.**IsNaN**(**accuracy**)** **||** float**.**IsInfinity**(**accuracy**))**

accuracy **=** 0.0f**;**

float precision **=** **(**float**)**tp **/** **(**tp **+** fp**);**

**if** **(**float**.**IsNaN**(**precision**)** **||** float**.**IsInfinity**(**precision**))**

precision **=** 0.0f**;**

float recall **=** **(**float**)**tp **/** **(**tp **+** fn**);** // true positive rate

**if** **(**float**.**IsNaN**(**recall**)** **||** float**.**IsInfinity**(**recall**))**

recall **=** 0.0f**;**

float fmeasure **=** **(**float**)** **((**fbeta **\*** fbeta **+** 1**)** **\*** precision **\*** recall**)** **/**

**((**fbeta **\*** fbeta**)** **\*** precision **+** recall**);**

**if** **(**float**.**IsNaN**(**fmeasure**)** **||** float**.**IsInfinity**(**fmeasure**))**

fmeasure **=** 0.0f**;**

float specificity **=** **(**float**)**tn **/** **(**tn **+** fp**);** // true negative rate

**if** **(**float**.**IsNaN**(**specificity**)** **||** float**.**IsInfinity**(**specificity**))**

specificity **=** 0.0f**;**

finalMatrix**.**Add**(new** List**<**float**>()** **{** accuracy**,** precision**,**

recall**,** fmeasure**,** specificity **});**

**}**

**}**

Se declara un set ClassTags pentru a putea avea fiecare clasa unica din setul de testare, pentru a le putea itera sa formeze matricia de confuzie pentru fiecare tag. In sectiunea sublinitata, se poate remarca pentru ce conditie cresc cele 4 valori, true negative creste atunci cand nici setul de test si tag-ul predictionat nu sunt aceleasi cu tag-ul curent din iteratia setului ClassTags, true positive creste cand si setul de test si tag-ul predictionat sunt egale cu tag-ul curent, false negative creste doar cand setul de test este acelasi cu tag-ul actual iar false positive creste doar cand clasa predictionata este aceasi cu tag-ul curent.

Dupa ce s-au realizat aceste metrici, rezultatele pot fi afisate pe ecran pentru a putea fi vizualizata performanta modelului si acuratetea acestuia.

# IV. Rezultate

# V. Anexa 1: Exemplu etichetare parte de vorbire

# VI. Bibliografie

[1] Jurafsky slp3-POST (2019): <https://web.stanford.edu/~jurafsky/slp3/8.pdf>

[2] TnT- A statistical POST (2000): <http://www.coli.uni-saarland.de/~thorsten/publications/Brants-ANLP00.pdf>

[3] Viterbi Algorithm wiki - <https://en.wikipedia.org/wiki/Viterbi_algorithm>

[4] Part of speech Tagging lecture 4 (2013) - <http://ivan-titov.org/teaching/nlmi-15/lecture-4.pdf>

[5] Part of speech Tagging lecture 5 (2013) - <https://staff.fnwi.uva.nl/k.simaan/D-Courses2013/D-NLMI2013/college5.pdf>

[6] Proiect POST pe github : <https://github.com/ST4NSB/part-of-speech-tagging>

[7] Brown Corpus manual: <http://korpus.uib.no/icame/manuals/BROWN/INDEX.HTM>

[] LINQ docs: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>

[] POS Implications of Affixes (1966) - <https://pdfs.semanticscholar.org/7008/6ddca220c59a215e815da69205bca2022158.pdf>

[] Grammar & Structure : prefixes & suffixes - <https://web2.uvcs.uvic.ca/courses/elc/sample/beginner/gs/gs_55_1.htm>

[] ESL Library: Suffixes that show the POS (2016): <https://esllibrary.com/blog/english-word-endings-suffixes-that-show-the-part-of-speech/>

[] Additive smoothing – wiki: <https://en.wikipedia.org/wiki/Additive_smoothing>

Brown Corpus wiki: <https://en.wikipedia.org/wiki/Brown_Corpus>

Brown Corpus data-set download: <https://archive.org/details/BrownCorpus>

Cross-Validation informatii: <https://en.wikipedia.org/wiki/Cross-validation_(statistics)>

Quantinsi Cross-validation exemple: <https://blog.quantinsti.com/cross-validation-machine-learning-trading-models/>

Cross-Validation exemple implementari: <https://scikit-learn.org/stable/modules/cross_validation.html>

Fisher-Yates Shuffling algoritm: <https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle>

Fisher-Yates Shuffling solutions (2009): <https://stackoverflow.com/questions/273313/randomize-a-listt>

File Reader in c# informatii: <https://docs.microsoft.com/en-us/dotnet/standard/io/how-to-read-text-from-a-file>

Tokenization (Analiza lexicala) wiki: <https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization>

Parti de vorbire in gramatica limbii engleze (1): <https://www.grammar.cl/english/parts-of-speech.htm>

Parti de vorbire in gramatica limbii engleze (2): <http://www.butte.edu/departments/cas/tipsheets/grammar/parts_of_speech.html>

Parti de vorbire in gramatica limbii engleze (3): <https://www.englishclub.com/grammar/parts-of-speech.htm>

Parti de vorbire in gramatica limbii engleze (4): <https://www.english-grammar-revolution.com/parts-of-speech.html>

[] Task Class in .NET (System.Threading): <https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=netcore-3.1>

[] Confusion matrix - wiki: <https://en.wikipedia.org/wiki/Confusion_matrix>

[] Curs Data mining | Analiza si evaluarea datelor – Daniel Morariu (2019)