**Coperta**

Cuprins

[I. Prezentare temă 4](#_Toc42386288)

[II. Considerații teoretice – algoritmi de învățare 4](#_Toc42386289)

[III. Consideraţii practice 7](#_Toc42386290)

[3.1 Descriere generală proiect 7](#_Toc42386291)

[3.2 Dezvoltare aplicație 8](#_Toc42386292)

[3.2.1 Dataset 8](#_Toc42386293)

[3.2.1.1 Împărțire 70% - setul de antrenament, 30% - setul de testare 10](#_Toc42386294)

[3.2.1.2 Cross-Validation 10](#_Toc42386295)

[3.2.2 Preprocessing 13](#_Toc42386296)

[3.2.2.1 Sentence tokenization 13](#_Toc42386297)

[3.2.2.2 Part of speech classifier 14](#_Toc42386298)

[3.2.2.3 Data cleaning & normalization 17](#_Toc42386299)

[3.2.3 Model 18](#_Toc42386300)

[3.2.3.1 Hidden Markov Model 18](#_Toc42386301)

[a) Emission probability 18](#_Toc42386302)

[b) Transition probability 21](#_Toc42386303)

[3.2.2.1 Unknown words model 28](#_Toc42386304)

[3.2.4 Decoder 38](#_Toc42386305)

[3.2.4.1 Forward method 40](#_Toc42386306)

[3.2.4.2 Backward method 46](#_Toc42386307)

[3.2.4.3 Bidirectional method 47](#_Toc42386308)

[3.2.5 Evaluation 49](#_Toc42386309)

[3.2.5.1 General Accuracy 49](#_Toc42386310)

[3.2.5.2 Confusion matrix 50](#_Toc42386311)

[IV. Rezultate 54](#_Toc42386312)

[V. Bibliografie 55](#_Toc42386313)

# 

# I. Prezentare temă

# II. Considerații teoretice – algoritmi de învățare

* Vb natural language processing
* Part of speech: rulebased, stochastic methods
* Algoritmit folositi in pos
* - viterbi, bigram, trigram
* < 20 pag

Ecuatia cu decoder produs tranzitie \* emisie

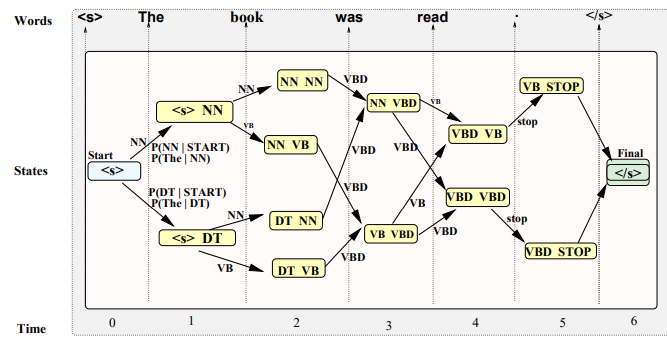
****

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.

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 Dezvoltare aplicație

### 3.2.1 Dataset

Setul de date (numit și text corpus) este o colecție de date de tip text prelucrate și alese special pentru a putea evalua calitatea sistemului de etichetare. Pentru setul de date se va folosi **Brown Corpus**, o colecție de propoziții și fraze în limba engleză colectate și organizate de W. Nelson Francis & Henry Kucera din departamentul lingvistic de la Universitatea Brown. Colecția aceasta are peste 1 milion de cuvinte în total și conține exact 500 de documente [7]. Cele 500 de documente sunt împărțite în 2 mari categorii, prima categorie fiind proză informativă cu următoarele subcategorii:

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 proză imaginativă 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 cuvânt (numit si token) este delimitat de un slash ‘/’, urmat de tagul aferent părții de vorbire al acestuia, sub forma “token/tag”. Setul de taguri folosit are numele de **Penn Treebank**, iar acesta conține 45 de taguri:



Figura 3.2 – Penn Treebank tagset [1]

Acest set de date este folositor pentru un algoritm de *invățare supervizată pe o clasă* sau *one class classification*. Exemplu de propoziție 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 ./.*

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

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

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

Alte taguri pot apărea formate din tagul propriu-zis și un tag prefix de indicație a unei informații suplimentare folosind delimitatorul ‘-‘, că fie este un cuvânt preluat din altă limba ‘FW’ (foreign word), fie că acel cuvânt apare ca titlu ‘TL’ in corpus, etc.

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

*,* unde */fw-in, /fw-nn* sunt tagurile pentru prepoziție, respectiv substantiv la singular, într-o limbă străină.

În unele taguri pot apărea și simboluri precum ‘$’(cuvânt la posesiv), ‘\*’(cuvinte negate precum can’t, wouldn’t, shouldn’t), ‘nil’(nespecificat).

**Train & test set**

Înainte ca modelul de predicție să fie folosit pe date reale (fined-tuned model), acesta mai întâi este evaluat pe setul de date cunoscut. Pentru a realiza evaluarea modelului, mai întâi trebuie să împărțim setul de date într-un set de antrenament (train set) și un set de testare (test set). Există 2 metode de împărțire a setului de date implementate în proiect:

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

Această metodă este oarecum evidentă, pentru fiecare subcategorie din Brown Corpus, se aleg din documentele aferente acesteia, 70% documente pentru etapa de antrenare și 30% documente pentru etapa de testare.

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

#### 3.2.1.2 Cross-Validation

Numită și “rotation estimation”, este o tehnică 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 predicția datelor noi, înlăturând probleme precum “overfitting” sau “selection-bias”, probleme în care procesul de învățare eșuează să facă predicții pe date noi deoarece modelul nu este generalizat pentru posibilități de date necunoscute.

**K folds cross-validation**

Acesta presupune mai întâi alegerea unui număr K (deobicei K = 4 sau K = 10) și apoi împărțirea setului de date în K părți numite folds. După această împărțire, se vor itera aceste folduri, astfel încât fiecare fold să fie la rândul lui set de testare iar restul foldurilor vor deveni set de antrenare. După fiecare evaluare per fold, se va salva acuratețea modelului pentru fiecare fold în parte, până la final când vom evalua acuratețea pe toate foldurile [8].



Figura 3.3 – exemplu cross-validation, k = 4 [9]

La final, după ce am evaluat fiecare fold în parte, se va calcula media aritmetică pe folduri, astfel încât să obținem o acuratețe pentru tot setul de date.

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

k - numărul de folduri alese

– acuratețea pentru foldul i

**Implementare K folds cross-validation**

Funcția principală pentru cross-validation are 3 parametrii de intrare: *filePath* pentru path-ul la folder-ul cu documentele Brown Corpus, *fold* pentru numărul de fold-uri folosite pentru împărțirea fisierelor (parametru default = 4) și *shuffle* pentru opțiunea de amestecare a documentelor (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**;**

**}**

**}**

În funcția aceasta, se vor citi toate documentele din folderul cu fișierele Brown și se vor returna textul acestora ca listă pentru fiecare document separat (files.count = 500).

În continuare, se vor iniția vectorii pentru fișierele de antrenare si de testare, se va calcula câte fișiere sunt per fold (de exemplu pentru fold = 4, vor exista 125 (500/4) documente per fold) iar după aceea, se va verifica dacă opțiunea de shuffle a fost “activată”, dacă da, atunci lista de fișiere va fi amestecată. În for-ul din funcția principală, se va itera fiecare fold în parte și se va calcula lista individuală de fișiere pentru etapa de antrenare și de testare. După aceasta, se vor concatena listele într-un singur string și se vor salva într-un array la index-ul foldului de test calculat.

Pentru a înțelege mai bine condiția sublinitată, se ia ca exemplu filesPerFold = 125, crossIndex = 3 (adica ultimul fold de calculat), atunci această conditie s-ar traduce => i(indexul pentru document) mai mare sau egal ca 375 și i mai mic strict ca 500, ceea ce reprezintă 25% pentru setul de testare salvat în lista *IndividualTestFiles* iar celelalte fișiere de la 0 la 374, adică restul de 75% se vor salva în lista de antrenare *IndividualTrainFiles*.

Funcția de amestecare implementează algoritmul *Fisher–Yates Shuffling,* acesta alege un număr random între indexul de la primul element din listă și ultimul element din listă după care începe să numere până la acesta, ignorând elementele din listă deja folosite pentru amestecare (scratched elements), odată ajuns la indexul respectiv, va prelua elementul de la acel index și îl va adăuga la finalul listei nou formate, punând astfel un flag de folosit pentru indexul respectiv si decrementând numărul total de numere nefolosite [10].

Totuși, implementarea acestuia este destul de ineficientă si complicată, pentru algoritm ar trebuii folosite 2 liste și un flag pentru indexul elementelor care au fost deja alese dar pentru a depăși aceste probleme, există o metodă mai ușoară de implementare, anume metoda lui *Durstenfeld's.* Aceasta presupune interschimbarea elementului de la indexul n (inițiat cu valoarea listei la început), cu elementul de la indexul ales random, decrementând n după fiecare interschimbare.

ex. versiunea lui Durstenfeld’s: 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]

Algoritmul în versiunea lui Durstenfeld este următorul:

**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 în care datele sunt curățate și normalizate, astfel încat lucrul cu acestea să fie unul corect si curat. Lipsa acestui proces poate îngreuna procesul de învățare si poate influenta negativ calitatea rezultatelor. Cele 3 procese implementate 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 și posibil a le clasifica, folosit în analiza lexicală. Acesta deobicei delimitează pe baza unei reguli, algoritmul folosit pentru setul de date este *Whitespace Tokenizer*.

Algoritmul *Whitespace Tokenizer* împarte un string într-o singură listă de stringuri, unde fiecare string este delimitat și adăugat în listă atunci când se întâlnește caracterul de spațiu, 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 implementat în proiect este echivalent cu funcția split din c#. Acest tokenizer este cel mai potrivit deoarece, în setul de date, fiecare cuvânt+tag este delimitat de spațiu, deci lista de tokenuri va arăta de forma: list = […., “he/pn”, “made/vbd”, “pancakes/nn”, “./.” , ….].

Pentru a putea lucra cu tokenul si tagul separat, se preferă despărțirea cuvântului 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**;**

**}**

**}**

Cuvântul este separat de tag prin funcția split cu parametrul de slash ‘/’ și se va obține un vector de stringuri. Ultimul string din vector va fi mereu tagul cuvântului dar pentru restul vectorului nu există aceeași siguranță, de exemplu cuvintele compuse precum “input/output” pot apărea în setul de date și dacă acestea nu sunt salvate integral, pot încurca procesul de învățare.

Se cunoaște faptul că în cazul cel mai bun, există un vector de stringuri de dimensiune = 2 (cuvânt necompus+tag) iar în cazul opus, se iterează vectorul de cuvinte și se concatenează tot ce apare în vector până la tag, după care se șterge ultimul caracter(care este caracterul de slash ‘/’) din stringul obținut, deoarece după fiecare cuvânt se adaugă caracterul de slash ‘/’. La final, se va adăuga în lista de structuri, atât stringul ce conține tokenul compus sau cel necompus și stringul ce conține tagul acestui token.

#### 3.2.2.2 Part of speech classifier

Algoritmului de calcul i-ar lua foarte mult să calculeze probabilitățile tuturor tagurilor (aproximativ 100 de taguri) dacă modelul s-ar aplica pentru fiecare tag în parte iar predicția ar avea de suferit în acest caz. Pentru a rezolva această problema, se alege clasificarea părților de vorbire în 10 categorii, acestea fiind părțile de vorbire de bază pentru a acoperi majoritatea tagurilor din Brown Corpus. Acest proces nu este unul automat făcut de un algoritm, ci este făcut în urmă unei analize a părților de vorbire din engleză [11], [12], [13], [14] și alese cele mai potrivite pentru setul de date cu care se lucrează (Brown Corpus). S-au ales următoarele categorii:

1. Noun (NN) – substantive
2. Pronoun (PN) - pronume
3. Verb (VB) – verb
4. Adjective (JJ) – adjectiv
5. Adverb (RB) – adverb
6. Preposition (PP) – prepoziție
7. Conjuction (CC) – conjuncție
8. Article/Determiner (AT/DT) – articole & determinanți (din limba engleză)
9. End of sentence (.) – sfârșit de propoziție sau marcarea unei început de propoziție
10. Others (OT) – alte părți de vorbire cum ar fi interjecție, cardinal numeral (“six”, “two”), cuvinte din altă limba, etc.

Orice cuvânt din setul de date va avea doar un singur tag din cele 10 menționate anterior, după ce va fi aplicat acest proces. În urma clasificării, obținem următoarele rezultate statistice:



Figura 3.4 – Frecvenţa noilor taguri



Figura 3.5 – Procentul pentru fiecare tag (%)

Se observă că aproape un sfert de cuvinte din setul de date sunt substantive, dacă s-ar creea un etichetator cu parametru implicit “substantiv”, s-ar putea predicționa corect 23.56% de cuvinte din setul de date.

**Implementare part of speech classifier**

Procedeul principal este de a itera vechea listă de cuvinte și de taguri și de a creea o nouă listă în care vechile taguri vor face parte dintr-una din cele 10 categorii. Pentru a găsi indexul la tag în listă avem următorul 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 returnează un index în funcție de unde se află tag-ul în lista BrownCorpusTags, mai întâi va despărți tagurile compuse și va itera fiecare tag în parte pentru a îi atribuii un index în listă. Se va verifica dacă există tag-ul iterat în lista BrownCorpusTags, dacă există se va returna index-ul din lista, dacă nu, se va returna -1. Metoda caută tagul cu funcția Contains deoarece unele taguri precum “nps$” apar și cu simboluri dar aparțin tot in aceeași categorie unde este “nps” catalogat.

Lista BrownCorpusTags conține următoarele taguri:

"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"**,** "."

, tagurile din această listă sunt ordonate în funcție de categoriile descrise anterior. Condiția sublinitată anterior este un caz special deoarece, funcția găsește secvența cu pattern-ul “ql” înainte de a găși “wql” în listă (“ql” este catalogat ca adverb iar dacă va cauta cu funcția Contains atunci “wql” va fi catalogat ca adverb când ar trebuii să fie conjuncție) și de accea se verifică mai întâi dacă tagul este egal cu “wql” sau “wql-tl” (aici nu apar taguri compuse cu ‘+’ sau cu taguri “fw”, “hl” etc.). Tagurile care nu apar în listă vor avea index = -1, și vor fi catalogate cu tagul de Others (altele).

Condiția de clasificare ConvertBrownTagToHierarchicTag, care în funcție de indexul returnat de metoda anterioară va decide noul tag pentru fiecare cuvânt este:

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**;**

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 fiecărui cuvânt printr-un filtru de curățare și de normalizare pentru a elimina cuvintele nepotrivite și pentru a normaliza datele astfel încât să nu încurce algoritmul de învățare.

Acest algoritm va elimina mai întâi cuvintele de oprire (caracterele care nu sunt importante la etichetarea părții de vorbire), acestea fiind: parantezele rotunde ‘()’, parantezele pătrate ‘[]’ și acoladele ‘{}’. După această etapă, algoritmul verifică dacă tokenul este un număr, dacă conține doar cifre atunci îl elimină, dacă conține și cifre dar și litere atunci elimina cifrele din acesta și dacă trece un anume prag de litere rămase atunci îl păstrează. Înainte de a termina, acesta, in etapa de antrenare, va păstra și o listă de cuvinte  care încep doar cu literă mare și încă o listă în care toate cuvintele sunt normalizate la literă mică. Cuvintele din setul de testare vor trece prin același filtru, cu excepția ultimei etape, acestea nu vor fi normalizate la literă mică.

  Acest procedeu de normalizare ajută la creșterea acurateții, deoarece etichetatorul va putea să facă distincția corectă între partea de vorbire a cuvintelor cu literă mare și cele cu litera mică. În setul de testare, de asemenea, se vor elimina repetițiile la caracterele cu tag de final de propoziție, deoarece acestea nu sunt evaluate de predictor și repetițiile de genul *“ ?/. ./. ?/. !/.”* pot emite erori dacă nu sunt tratate. Alte funcții de normalizare și de curățare folosite de celelalte componente sunt descrise în capitolele următoare.

**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**;**

**}**

, implementarea funcției care elimină cifrele:

**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**;**

**}**

**}**

Cu ajutorul clasei **Regex**, în secțiunea subliniată, se înlocuiește locul unde apare o cifra (d = digit) cu un caracter gol (adică elimină cifra). Funcția ToLowerCaseNormalization(string) este echivalentă cu funcția ToLower() din c#.

### 3.2.3 Model

Modelul este partea principală din sistem, acesta conține diferite informații despre ponderile și probabilitățile 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 stohastic (folosește metode probabilistice pentru a predicționa tagurile) și este implementat după modelul clasic prezentat. Acesta folosește probabilitățile de emisie și de tranziție interpolate, pentru a predicționa tagul cuvintelor din propoziție.

##### Emission probability

Probabilitatea de emisie (numită și likelihood) se calculează dat fiind un tag, care este probabilitatea de asociere al acestuia cu un cuvânt dat.

|  |  |
| --- | --- |
|  | (3.2) |
| Unde, probabilitatea de emisie a cuvântului asociat cu tagul  frecvența de apariție a cuvântului asociat cu tagul  frecvența de apariție a tagului | | |
| ex. … ***he/???***went/VB home/NN … | (3.3) |

Putem vedea în exemplul anterior cum această formulă este o generalizare de la formulă naivă a lui Bayes, important de remarcat că aceasta nu incearcă să răspundă la întrebarea: “care este cel mai probabil tag pentru cuvântul ‘he’?”, ci “dacă am genera un pronume, cât de probabil ar fi ca acesta să fie ‘he’?’ [1]. Valorile din exemplul anterior sunt alese aleator pentru demonstrație.

**Implementare emission probability**

Pentru a putea fi aplicată formula de la (3.2), mai întâi trebuie obținută frecvența de apariție a tuturor tagurilor din setul de antrenare iar după aceea, se obține frecvența de apariție a tuturor cuvintelor împreună cu fiecare tag asociat acestuia. Pentru a păstra fiecare asociere cu fiecare tag la un token, se va creea o listă 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**;**

**}**

**}**

S-a menționat la subcapitolul anterior *Data cleaning & normalization* că se va ține o listă separată pentru cuvintele care încep cu literă mare și toate cuvintele normalizate la literă mică, codul care adună frecvența de apariție în lista pentru cuvintele cu litere mari este următorul:

**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 aplică sintaxa **LINQ** [15] pentru a căuta dacă în lista cu frecvențe avem deja cuvântul pe care îl iterăm (Find(x => x.Word == w.word)), dacă nu găsim cuvântul în listă atunci se creează un obiect de tip EmissionModel și îi se adaugă cuvântul și tagul aferent acestuia cu un count = 1, după care este adăugat în lista WordCapitalizedTagsEmissionFrequence. Metoda LINQ FristOrDefault va returna primul obiect din listă/dicționar care îndeplinește condiția expresiei lambda sau null/0 în cazul în care niciun obiect nu îndeplinește condiția. În cazul în care cuvântul este deja în listă, atunci se caută tagul aferent al acestuia , dacă nu se găsește tagul atunci se adaugă noul tag la dicționar, altfel dacă acel tag căutat există, atunci se incrementează frecvența de apariție a tagului de la cheia găsită. Același procedeu se aplică si pentru cuvintele normalizate la literă mică.

Odată ce s-a calculat frecvența de apariție a tuturor tagurilor fiecărui cuvânt, se poate aplică  formulă (3.2) doar pentru cuvintele care există în setul de testare. De remarcat că nu se calculează probabilitatea tututoror cuvintelor din setul de antrenare, ci doar acelora care sunt întâlnite în setul de testare, aplicând funcția de calculare a probabilității doar pentru setul de testare. Deoarece există 2 liste care dețin informații legate de frecvențele de apariție, și probabilitatea de emisie, pentru setul de testare, se va calcula și se va memora în 2 liste. Clasa care reține informațiile legate de probabilitățile de emisie este:

**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 care implementează formula de la (3.2) pentru cuvintele normalizate la literă mică este:

**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**);**

**}**

**}**

În condiția evidențiată, se verifică dacă există cuvântul curent din setul de testare în lista antrenată și se mai verifică tot pentru acesta dacă nu este în lista finală cu probabilitățile de emisie. În cazul în care cuvântul apare pentru prima dată și se află în lista de antrenare, atunci se continuă procesul de iterație pentru toate tagurile al acestui cuvând, și se calculeaza probabilitatea de emisie pentru fiecare. Variabila cti va căuta tagul respectiv în lista cu frecventele unigramului (această fiind descrisă mai tarziu la sectiunea *Bigram & unigram*), după care se poate aplica în sfârșit formula de la (3.2), unde tf.Value este frecvența de apariție a tagului actual asociat cuvântului din setul de testare, pe (supra) frecvența de apariție a unigramului calculată anterior.

Pentru cuvintele care încep cu literă mare, se va aplica același proces, doar că rezultatele probabilității se vor păstra într-o lista diferită (WordCapitalizedTagsEmissionProbabilities) și în loc de prima linie din codul anterior:

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

, se va verifica dacă în schimb cuvântul începe cu literă mare:

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

##### Transition probability

Probabilitatea de tranziție (numită și prior probability) se calculează dat fiind un tag, care este probabilitatea de apariție după un anume tag dat.

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

Unde, probabilitatea bigramului pentru secvența de taguri

frecvența de apariție a secvenței bigram a tagurilor

frecvența de apariție a tagului

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

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

În exemplul dat, calculăm probabilitatea ca “likes” luat ca verb, să urmeze după un substantiv, adică numărul de apariții a unui substantiv (NN) urmat de un verb (VB) este 8027 pe (supra) numărul de apariții total a unui substantiv (NN) este 13038, rezultatul fiind  0.61 în exemplul demonstrativ anterior.

**Bigram & unigram**

Formula dată la (3.4) este formula de calculare a probabilității unui bigram (sau 2-gram), deoarece este formată din numărul de apariții a 2 taguri urmat unul după altul. Formula care calculează probabilitatea unui singur tag, numită și unigram (1-gram), este următoarea:

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

Unde, probabilitatea unigramului pentru tagul

frecvența de apariție a tagului

N – numărul de tokeni (cuvinte) din setul de antrenare

Cu formulele (3.2), (3.4) și (3.6) se poate creea un sistem de predicție bazat pe Hidden Markov Model dar numai cu acestea nu se obțin cele mai bune rezultate.

**Implementare Bigram & unigram**

Pentru a se putea mplementa formulele de la (3.5) și (3.6) mai întâi trebuie obținute frecvențele de apariție a tagului individual (unigram) și a 2 taguri unul după altul (bigram). Pentru asta este  nevoie de un dicționar care să țină o evidență a tuturor tranzițiilor:

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

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

Codul pentru a număra secvențele 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 că în condiția subliniată s-a pus un flag de verificare, în acesta se intră la prima iterație a buclei for. Se realizează această verificare pentru prima tranziție din setul de antrenare, (NULL, NN) echivalentă cu (“.”, NN) deoarece și începutul de propoziție este un  tag  valid din cauză că se poate obține probabilitatea de tranziție bigram pentru tagul primului cuvânt din propoziție. Cum primul cuvânt din setul de antrenare nu are un flux de date înaintea lui, atunci această verificare este necesară. Pentru a păstra tranzițiile părților de vorbire, s-a ales ca tip de date un tuplu cu 2 parametrii, primul parametru fiind tagul de dinainte (precedentul) și al doilea parametru fiind tagul care urmează după precedent (tagul curent). Implementarea este foarte asemănătoare la concept cu implementarea din probabilitatea de emisie.

Pentru unigram, implementarea este următoarea:

**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**;**

**}**

**}**

La secțiunea anterioară *Emission probability,* a apărut utilizarea acestei metode AddTagToUnigramOccurences(..),  în loc de a itera de 2 ori setul de antrenare, se iterează doar o singură dată și se adaugă în aceelas timp și frecvențele de apariție a tuturor cuvintelor cu taguri (word-tag emission count) și frecvențele de apariție a tuturor tagurilor individuale (unigram count).

Diferența față de calcularea probabilității pentru lista de emisie și dicționarele de unigram și bigram,  este aceea că toate tuplurile din dicționarele de unigram și bigram vor fi calculate indiferent ca probabilități deoarece acestea sunt  *legăturile* (numite și arce într-un graf orientat) *stărilor ascunse*(noduri) și nu sunt vizibile în setul de testare. Pentru a putea ști câte legături pot exista în total la un n-gram, se dă următoarea formulă:

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

unde, T – numărul total de legături posibile într-un n-gram

x – numărul tuturor părților de vorbire individuale din corpus

n – n-gramul ales (1 pentru cazul unigram, 2 pentru cazul bigram, 3 pentru cazul trigram, etc.)

ex. n = 2 (bigram) si x = 10 (10 taguri diferite),

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

Într-un bigram cu 10 părți de vorbire diferite, există în total 100 de legături pentru aceste stări ascunse ([NN,VB], [NN,NN], [CC,PN], etc.).

Funcțiile de calcul a probabilității unigramului și a bigramului sunt următoarele:

**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**);**

**}**

**}**

Fracțiile probabilității de tranziție au fost scăzute cu o constantă = 1 și la numărător și la numitor, la secțiunea *Trigram & bigram smoothing* este explicat motivul din spatele acestei scăderi. Rezultatele vor fi memorate în dicționare de tip double (probabilități):

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

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

**Trigram**

În multe cazuri, uitându-ne doar la cuvântul de dinainte nu este destul, avem nevoie de mai multă informație, știind întreg contextul e mai util decât să cunoaștem doar cuvântul de dinainte. Pentru asta, se introduce trigramul, acesta se uită înapoi la ultimele 2 cuvinte, față de bigram care se uită doar la cuvântul de dinainte. Pentru a calcula probabilitatea trigramului se dă formula:

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

Unde, probabilitatea trigramului a secvenței de taguri

frecvența de apariție a secvenței trigram ale tagurilor

frecvența de apariție a secvenței bigram a tagurilor

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

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

În exemplul anterior, se calculează ca, dat fiind bigramul cu cele 2 taguri precedente (DT-determinant, JJ-adjectiv), care este probabilitatea de apariție a unui substantiv (NN). Trigramul este cel mai potrivit n-gram pentru etichetarea părții de vorbire pentru un model de tip Hidden Markov, returnează cea mai bună acuratețe dintre cele trei (unigram, bigram, trigram) dar este și cel mai intensiv din punct de vedere al timpului computațional.

**Implementare trigram**

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

Frecvență apariții:

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

Probabilități:

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

Algoritmul implementat de numărare a frecvențelor de apariție:

**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 asemănător cu cel descris la bigram, diferența majoră fiind evidențiată în a două condiție, aici nu se mai salvează și secvențele unde este întâlnit sfârșit de propoziție(sau început de propoziție) în mijlocul tuplului. Acest lucru este realizat deoarece, propozițiile/frazele sunt de sine stătătoare, partea de vorbire a unui token de început nu depinde de tagul ultimului token din propoziția anterioară. Chiar dacă asta ar putea ajuta sistemul la o acuratețe mai bună (multe propoziții se termină cu un substantiv/verb și încep cu un substantiv sau articol), nu ar fi util în aplicațiile reale, deoarece de multe ori datele reale sunt introduse de userii care folosesc aplicația (într-o cantitate mult mai mică) ci nu sunt preluate dintr-un set mare de date.

Funcția de calcul a probabilității trigramului descrisă în formula (3.9) este:

**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**

Probabilitățile de tip trigram generate dintr-un corpus *deobicei* nu pot fi folosite direct din cauza valorilor lipsă a tuplurilor trigram (sparse-data problem [2]). Pentru a rezolva această problema se introduce conceptul de interpolare liniară. Aceasta presupune calcularea unei noi probabilități compuse din suma probabilităților de tranziție înmulțite cu o pondere:

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

Valorile sunt estimate prin *interpolarea eliminată*. Pseudocodul general de determinare a acestor ponderi este următorul:



Figura 3.6 – pseudocodul pentru interpolarea liniară (Deleted Interpolation) a trigramului [2]

Se poate observa cum probabilitățile fiecărui n-gram sunt scăzute de constanta 1 și la numărător și la numitor, asta înseamnă că algoritmul ia în calcul date care nu au apărut în setul de antrenare. Cu toate acestea menționate, interpolarea liniară nu este folositoare numai pentru a trata tupluri lipsă dar și pentru a seta ponderi fiecărui n-gram separat. Folosind această funcție, acuratețea predictorului crește, în cazul meu, creșterea acuratatii este mică deoarece setul de antrenare este clasificat în 10 clase (taguri) reprezentative și de aceea dicționarele bigram și trigram au aproape toate permutările posibile. Folosind formula de la (3.7) se poate verifica dacă n-gramul conține toate permutările sale posibile, astfel trăgând concluzia dacă interpolarea liniară aduce îmbunătățiri majore la predicție.

Înafară de interpolarea pe trigram, s-a realizat și o interpolare pe bigram, aceasta este foarte asemănătoare cu cea de la trigram, ea este prezentată mai departe la implemetare.

**Implementare deleted interpolation**

Algoritmul interpolării liniare pe 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 că valorile sunt luate din dicționarele aferente fiecărui n-gram și nu sunt recalculate pe loc. Pentru a se putea scădea 1 și la numitor și la numărător, s-a realizat această operație de scădere la calcularea finală a probabilității fiecărei tranziții descrisă în subcapitolul anterior. Valorile ponderilor lambda se vor păstra în membrii publici al clasei Model, fiind accesibile mai târziu decodorului pentru a putea aplica formula de la (3.11).

Algoritmul interpolării liniare pe 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**;**

După cum se poate vedea, algoritmul este aproape identic cu cel de la trigram, doar că aici nu există 3 ponderi, ci doar 2 ponderi, se iterează valorile de bigram și acestea vor fi valorile adunate pentru ponderi. Ponderile vor fi și ele accesibile decodorului.

#### 3.2.2.1 Unknown words model

În subcapitolul anterior am prezentat întreg modelul Markov, folosind doar acest model,  sistemul de etichetare poate obține o acuratețe destul de bună dar un sistem foarte bun de etichetare trebuie să poată trata și cuvintele care nu se găsesc în setul de antrenare. Pentru asta, există mai multe modalități de tratare precum: sistem bazat pe reguli, învățare nesupravegheată, algorimti care analizează structura cuvantantului etc. În această lucrare, se va prezenta modelul pentru tratarea cuvintelor necunoscute bazat pe două părți, una de analiză a sufixelor/prefixelor cuvântului și o parte bazată pe reguli implementate manual în urma unei analize a setului de date. Funcția finală care va combina aceste 2 părți, va primi 2 parametrii de intrare, acestea fiind cuvântul necunoscut și tagul cu care se va verifica asocierea cuvântului. Funcția va returna o probabilitate de asocierie a cuvântului cu tagul verificat. Pentru a putea folosi funcția, mai întâi va trebuii obținute informații pentru sufixe și prefixe din setul de antrenare.

**Prefix & suffix training phase**

Prefixul este un afix pus înaintea rădăcinii unui cuvânt, iar sufixul este pus după rădăcina cuvântului. Aceste 2 componente importante pot oferii informații despre partea de vorbire a unui cuvânt, în jurnalul “Wall Street”, cuvintele care se termină cu “able” sunt etichetate ca adjective în 98% din cazuri [2].

ex. incompatibility (substantiv)

În exemplul acesta, “în” este prefixul axifului “compatibili” iar “ity” este sufixul acestuia. De remarcat aici că și “ty” poate fi un sufix, atât cât și “y” (multe adverbe și adjective în engleză se termină cu caracterul “y”) dar sufixul întreg “ity” este cel mai specific pentru exemplul dat.

Pentru a se putea alege cele mai bune sufixe/prefixe, acestea nu au fost deduse și calculate din setul de antrenament (timp computațional mare și rezultate mediocre), ci au fost alese manual ca fiind cele mai reprezentative. În urma analizei din [16], [17], [18], prefixele și sufixele alese pentru acest 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" **};**

Ca sistemul să folosească aceste afixuri pentru a găsi tagul cuvintelor necunoscute, trebuie mai întâi să se observe cu ce tag vin asociate în setul de antrenament și să se calculeze probabilitatea de asociere cu fiecare tag.

Pentru a se putea calcula acest lucru, se folosește următoarea formulă:

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

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

– frecvența de apariție a sufixului/prefixului cu tagul

– constantă pentru realizarea netezirii **additive smoothing** [19]

– suma tuturor tagurilor asociate sufixului/prefixului

d – mărimea totală a setului de prefixe/sufixe

Se observă repetiția conceptulului de smoothing, acesta este foarte important aici deoarece se dorește obținerea unei probabilități diferite de 0 și pentru sufixele/prefixele care nu sunt întâlnite în setul de antrenament. În program, constanta fost inițiată cu valoarea = 1 , această metodă mai este denumită și **Laplace Smoothing**. [19]

ex.

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

Aceste calcule prezentate au fost realizate pentru sufixele/prefixele care încep cu literă mică dar în sistem este implementat aceelaș concept și pentru calculul sufixelor/prefixelor cuvintelor care încep cu literă mare deoarece aceste diferențe pot da rezultate mai bune la evaluarea sistemului.

**Implementare prefix & suffix training phase**

Pentru a putea calcula probabilitatea de la (3.12), este nevoie și de frecvențele de apariție a sufixului/prefixului cu fiecare tag. Algoritmul de contorizare a sufixelor si a prefixelor pentru cuvintele care încep cu literă mică:

**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**;**

**}**

**}**

**}**

**}**

De remarcat în această secțiune că ordinea sufixelor și a prefixelor, din lista implementării lor manuale prezentată anterior este foarte importantă, dacă se apelează metoda .EndsWith(..) și de exemplu sufixul “ty” se află înaintea sufixului “ity” atunci, sufixul “ity” nu va mai putea fi contorizat în lista de sufixe. Acest proces este exact la fel pentru sufixele și prefixele cuvintelor care încep cu literă mare, doar că pentru prefixe se trece cuvântul de la literă mare la literă mică cu metoda .ToLower().

Lista finală cu probabilități va fi una de tip EmissionProbabilisticModel, algoritmul de calculare a probabilității (3.12) pentru cuvinte care încep cu litera mică este următorul:

**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 încep cu literă mare procesul este exact la fel.

**Model training on separate threads**

În faza de creare și de antrenare a modelului, se crează listele importante cu informații despre probabilitățile de emisie, de tranziție și antrenarea sufixelor și a prefixelor. Calcularea acestora consumă timp dacă sunt calculate una după cealaltă, dar având în vedere faptul că aceste componente sunt separate și nu se influențează una pe cealaltă, acestea pot fi calculate asincron pe threaduri separate folosind clasa Task din .NET [20]. 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 declară un task pentru fiecare funcție, acesta fiind lansat la apelarea funcției .StartNew(), la final așteptându-se finalizarea tuturor operațiilor pentru a se putea ieși din metoda principală.

În faza de testare, când se calculează probabilitățile reale de emisie și de tranziție, se utilizează din nou clasa Task pentru a se calcula în paralel aceste probabilități:

**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**();**

**}**

**}**

**Rule-based tag weights**

Ultima componentă din funcția de recunoaștere a tagului pentru cuvinte necunoscute rămâne componenta bazată pe regulile impuse manual. Aceasta se bazează pe reguli precum: cuvintele care încep cu literă mare au o probabilitate mai mare să fie substantive, cele cu apostrof și care se termină cu ‘s’ au o probabilitate foarte mare să fie substantive, cuvintele care conțin cratimă (‘-‘) au o probabilitate mai mare să fie cuvinte compuse de tip OT (altele) sau JJ (adjectiv), etc. Pentru a seta la fiecare condiție o anumită pondere, s-au ales 2 parametrii care influențează ponderea finală pentru tagul care se verifică, 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 verifică la momentul intrării în funcția de recunoaștere a tagului pentru cuvântul (testWord) necunoscut, funcția de decodificare va incerca să testeze fiecare tag înafară de cel de sfârșit/început de propoziție, iar cel cu probabilitatea cea mai mare va fi ales ca tag pentru cuvăntul necunoscut.

**Unknown word final function**

Pentru a putea combina aceste componenente va trebuii să calculăm probabilitatea cuvântului necunoscut cu tagul curent în funcție de sufixele și prefixele asociate acestuia și  probabilitatea în funcție de condițiile trecute pentru ponderea de reguli. Acestea sunt combinate într-o probabilitate finală:

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

Unde, – probabilitatea cuvântului necunoscut asociat cu tagul

– formula de calcul a sufixelor și a prefixelor cuvântului necunoscut , asociat cu tagul

– formula de calcul pe baza regulilor trecute pentru cuvântul necunoscut , asociat cu tagul

Formula se calculează prin a aduna probabilitatea sufixului și a prefixului, dacă există prefix/sufix pentru cuvântul și tagul curent , mai apoi fiind normalizată cu formula min-max de la (3.15) în intervalul [0, 2]. Dacă nu este găsit sufix sau prefix, atunci va căuta cea mai mică valoare pentru sufix & prefix și va face același proces, dar dupa normalizare, va înmulți rezultatul și cu o constantă egală cu 0.01.

Formula se calculează prin a aduna ponderile pentru regulile care sunt trecute (occurenceAdder adună aceste ponderi, variabilă întâlnită anterior), este normalizată tot cu formula de la (3.15) dar cu limita superioară egală cu variabila bestValueWeight, adică intervalul [0, 2.5]. Pentru cazul când nu găsește nicio regulă și adunarea ponderilor este 0, atunci aceasta normalizează în intervalul [0, 2.5] valorea de la variabila worstValueWeight adică 1.5, mai apoi rezultatul fiind și el înmulțit cu aceași constantă egală cu 0.01.

La final când se vor aduna aceste componente rezultatul poate depăși intervalul probabilității, de aceea se execută o funcție care convertește valoare la maximul limitei superioare a probabilității adică 1.0. Acest lucru nu este greșit deoarece, dacă se trece de limita maximă de 1.0 atunci înseamnă că este oricum o probabilitate de 100% (siguranță totală) să fie tagul respectiv (deobicei această valoare este trecută pentru tagul de substantiv, care de cele mai multe ori este și tagul corect). Formula (3.14) este foarte importantă deoarece ea combină cele 2 formule astfel încât să se obțină o probabilitate chiar dacă se nu găsesc sufixe/prefixe sau nu este trecută nicio condiție dar să se poată obține și o probabilitate bună dacă nu găsește sufixe/prefixe dar în schimb sunt trecute câteva condiții sau invers dacă sunt găsite sufixe/prefixe dar nu se trece nicio condiție.

Formula normalizării min-max:

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

Unde, noua valoare

vechea valoare

min – limita inferioară

max – limita superioară

ex. 1. Cuvântul necunoscut = “romana” si tagul = “NN”

nu are niciun sufix sau prefix recunoscut în lista de sufixe/prefixe deci presupunem că suma valorii probabilității minime a sufixului și a prefixului este = 0.17 (sufix = 0.10, prefix = 0.7)

Folosind formula de la (3.15) rezultă:

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

Cuvântul “romana” nu trece nicio condiție impusă anterior, aplicând (3.15):

|  |  |
| --- | --- |
| Aplicând formula de la (3.14) se obține: | (3.18)  (3.19)  (3.20) |

Se poate observa cum probabilitatea este foarte mică dacă nu se trece nicio condiție și nu se găsește niciun sufix/prefix.

ex. 2. Cuvântul necunoscut = “Romanian” si tagul = “NN”

În acest caz, sufixul cuvântului este “ian”, luând probabilitatea din modelul oficial deja antrenat al aplicației, se obține suma = 0.239 (0.239 = sufix, 0.0 = prefix):

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

Aici nu se mai înmulțește cu 0.01 deoarece a fost deja găsit un sufix în setul de antrenare.

De asemenea, se trece o singură condiție, cea în care cuvântul începe cu literă mare, deci rezultă:

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

Cuvântul Romanian are o probabilitate foarte mare să fie asociat cu tagul de substantiv. Cu toate că din acest exemplu reiese că Romanian este cel mai probabil un substantiv (pentru cazul “The Romanian who won the nobel prize is ..” atunci funcția returnează rezultatul corect), în majoritatea cazurilor el este adjectiv (“My Romanian friends ..”) . Ca funcția să returneze cel mai bun rezultat, ea trebuie combinată cu o probabilitate de tranziție, acest lucru se va prezența în subcapitolul următor numit *Decoder*.

**Implementare Unknown words function**

Mai întâi se vor obține probabilitățile sufixelor și a prefixelor pentru cuvântul necunoscut și tagul curent iar după aceea se va calcula suma bazată pe ponderile regulilor pentru condițiile trecute. Primul pas este obținerea valorii minime atunci când nu este găsit un sufix sau un prefix pentru cuvântul dat, după care începe căutarea prefixelor/sufixelor cuvântului dat. Se începe căutarea sufixelor și a prefixelor cuvintelor care încep cu literă mare iar dacă nu sunt găsite acolo, sunt căutate în lista cu sufixe și prefixe pentru cuvintele doar cu literă mică.

Căutare 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**;**

**}**

**}**

Căutare sufix & prefix pentru cuvântul de intrare:

**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;**

**}**

**}**

**}**

După cum se poate observa, odată găsit sufixul/prefixul, se va ieși din iterație deoarece nu trebuie căutat un alt sufix/prefix care este mai abstract ca primul găsit. Metoda *ContainsKey*(..) verifică dacă există cheia cu tagul curent în sufixul/prefixul respectiv fără a mai trebuii iterat și dicționarul de taguri. Algoritmul pentru sufixele/prefixele pentru cuvintele care încep cu literă mică este identic.

După ce s-a finalizat acest calcul, se poate trece mai departe la calculul sumei și la normalizare pentru formula

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

**}**

Pentru formula bazată pe regulile și condițiile trecute, formula de calcul a ponderilor este implementată 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 și lowerAdderBound sunt egale cu bestValueWeight (2.5) respectiv worstValueWeight (1.5). Înainte de a se returna rezultatul, valoarea finală trebuie convertită în intervalul unei probabilități:

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

unde BoundProbability este implementată tot în 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**;**

**}**

### Decoder

În subcapitolul anterior am prezentat formarea modelului pentru cuvintele cunoscute și cuvintele necunoscute, probabilități de emisie și de tranziție. În acest subcapitol, se prezintă partea de decodor a modelului, fără acesta nu s-ar putea determina secvența variabilelor ascunse (secvența tagurilor) asociate cu secvența de observații (cuvintele unei propoziții) [1].

Un algoritm important de decodificare a unui model ascuns, pe baza programării dinamice, este algoritmul lui Viterbi. Algoritmul lui Viterbi poate procesa stările trellis-ului pornind de la stânga la dreapta dar de asemenea poate să o facă și invers. Am să numesc aceste metode forward (merge înainte de la primul cuvânt din propoziție până la sfârșitul propoziției), backward (merge de la sfârșitul propoziției la începutul acesteia) și bidirecțional (o combinație între ambele), acestea formează algoritmul de decodificare Viterbi. Formula generală de calculare a fiecărui nod este:

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

unde, probabilitatea nodului curent

probabilitatea nodului procesat la pasul de timp anterior

– probabilitatea de tranziție de la starea (tagul) anterioară la starea curentă

– probabilitatea de emisie (sau state observation likelihood) a observației simbolului (token) dându-se starea j curentă

Formula prezentată la (3.16) calculează probabilitatea maximă de trecere de la o parte de vorbire la alta, aceasta calculează probabilitatea dacă există o legătură bigram/trigram între tagul nodului anterior și tagul nodului curent, altfel nu necesită calcularea fiecărui nod de tag deoarece rezultatul ar fi 0.

Un nod Viterbi este implementat astfel în 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 folosește de obiectul PrevNode pentru a putea face backtack la nodul anterior, astfel pentru a putea decodifica întreaga secvență. NextNode se folosește pentru metoda de backward, iar ambele sunt folosite pentru metoda bidirecțională. Metodele nu calculează formula și pentru noduri unde rezultatul va fi zero (nu există probabilitate anterioară/emisie/tranziție).

Funcția care implementează algoritmul lui Viterbi este următoarea:

**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**);**

**}**

După cum se poate vedea, funcția primește 5 parametrii de intrare, primul este modelul în sine, al doilea este colecția setului de testare, al treilea și al patrulea specifică ce tip de tranziție se poate selecta (bigram/trigram) și ultimul specifică metoda de decodare (forward, backward, bidirecțional). Codul subliniat implementează metoda care șterge token-urile (cuvintele) care au tag de final/început de propoziție.

#### 3.2.4.1 Forward method

În această metodă, propozițiile sunt procesate de la stânga la dreapta, când se ajunge la nodul final cu valoarea cea mai mare, se face un backtrace pentru a returna etichetele finale. Metoda forward folosește formula de la (3.16), aceasta poate să folosească probabilitatea de tranziție bigram dar atât și trigram, totuși, pentru trigram aceasta nu va putea să calculeze nodul de început, de aceea, aceasta folosește probabilitatea de la bigram pentru a calcula probabilitatea primului nod din propoziție. Implementarea acestei metode și a celei backward este lungă și complicată, de aceea se vor prezența implementarea acestora pe bucăți de cod.

Metoda forward va inițializa un flag de start pentru propoziție nouă, un iterator pentru a recunoaște dacă se poate aplica trigram la nodul curent (în cazul în care modelul trigram este activat din funcție) și va itera de la 0 la N, unde N este ultimul token din setul de testare. La fiecare iterație se va verifica flag-ul de start, va crește iteratorul pentru trigram și se va obține probabilitatea de emisie pentru token-ul (cuvântul) modificat 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**());**

**......**

În cazul în care startPoint este true, atunci se cunoaște faptul că se verifică tranziția de la început de propoziție (tag-ul de ‘.’) la un alt tag care nu este început/sfârșit de propoziție și nu se poate aplica probabilitatea de tranziție a trigramului (ea poate fi aplicată în cazul în care triPoz >= 2). Mai întâi se verifică dacă există o tranziție bigram de trecere de la tagul de început de propoziție la tagul cuvântului găsit la timpul = i (foundWord), dacă găsește cuvântul în setul deja antrenat, atunci va itera prin tagurile acestuia, dacă nu îl găsește, atunci va itera prin toate probabilitățile de tranziție bigram de la tagul de început de propoziție la un alt tag diferit de acesta.

**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 când nu găsește cuvântul în lista cu probabilități de emisie, se va adăuga cuvântul necunoscut în lista de cuvinte necunoscute (se va folosi mai încolo la metricile de evaluare), se va itera lista cu probabilități de tranziție bigram și se va căuta ca al doilea tag să fie diferit de tagul început/sfârșit de propoziție. În momentul ce găsește un tag care respectă această condiție, algoritmul va calcula variabila biTrans ca adunare între probabilitatea de bigram (înmulțită la rândul ei cu ponderea lambda de bigram, calculată și explicată anterior la interpolarea liniară) și probabilitatea de unigram (la fel înmulțită cu ponderea  lambda de unigram). După aceasta, se va calcula proabilitatea cuvântului necunoscut (descrisă la capitolul anterior), cu tagul curent care va fi salvată în variabila unknownProcent. Valoarea finală este în sfârșit calculată ca produs din probabilitatea de tranziție și probabilitatea cuvântului necunoscut.

După ce această este calculată, se poate salva nodul cu valoarea probabilității pentru tagul curent, și tagul curent în lista cu noduri pentru iterația la timpul = i. Nodurile anterioare a primului cuvânt din propoziție va fi null.

Totuși, dacă se găsește cuvântul/token-ul în lista cu probabilități de emisie atunci se va execută următoarea condiție:

**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;**

Această secvență se va executa asemănător cu secvența descrisă anterior, doar că aici se vor itera doar tagurile care sunt asociate cuvântului actual, ci nu vor fi iterate toate tagurile posibile permise. Se poate observa că la produs, aici nu mai intervine probabilitatea cuvântului necunoscut ci doar probabilitatea de emisie pentru fiecare tag de emisie al cuvântului curent, înmulțit cu probabilitatea de tranziție descrisă anterior.

În continuare, după ce se calculează lista de noduri la timpul = i, se va adăuga în matricea 2d dinamică (lista de lista) ViterbiGraph și se va seta startPoint pe false deoarece acum a trecut de primul cuvânt din propoziție și se poate trece la condiția unde există și un nod anterior și se poate aplica trigram dacă funcția a fost apelată cu acest parametru pentru model. Pentru condiția când s-a ieșit din start, procesul este asemănător, se poate găsi cuvântul în lista cu probabilități de emisie antrenate sau poate să nu existe în acea listă, codul pentru cazul când nu se află in listă și modelul este selectat ca trigram este următorul:

**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**);**

**}**

**}**

**}**

În acest caz, mai întâi se va adăuga cuvântul necunoscut în listă, și se vor itera toate nodurile anterioare existente, pentru a se putea folosi formula de calcul a probabilității nodului curent (3.16). Chiar dacă se vor itera toate nodurile, doar un singur nod va face legătura cu nodul curent, acela fiind nodul cu cea mai mare probabilitate de transfer. Dacă poziția triPoz este egală cu 2 (adică se poate obține probabilitatea de trigram deoarece a trecut de secțiunea unde nu există trigram) și modelul de decodificare a fost ales trigramul, atunci condiția se va îndeplini și se va calcula probabilitatea pe nodul respectiv cu probabilitatea de tranziție de tip trigram. vGoodNode va fi nodul care va avea informații despre nodul maxim calculat, procesul de calculare fiind asemănător cu cel descris anterior doar că aici se va calcula variabila triTransition cu formula (3.11) descrisă la subcapitolul anterior. După ce s-a calculat probabilitatea de tranziție interpolată, se va calcula, asemănător anterior, probabilitatea cuvântului necunoscut pentru tagul curent.

Odată ce s-au calculat acestea, putem calcula probabilitatea nodului de la formula (3.16), înmulțind la cele menționate anterior și probabilitatea nodului anterior (adică nodul salvat în matricea dinamică 2d la linia *this.ViterbiGraph.Count – 1* și coloana curentă *j*. Dacă acest produs este mai mare ca vGoodNode, atunci se va înlocui valoarea veche a nodului curent cu valoarea calculată recent și se va salva și obiectul elem, acesta fiind nodul curent care la timpul = i+1 va deveni nodul precedent. Se poate observa că în condiția subliniată, se vor lua doar tranzițiile unde primul element din tuplul trigram este tagul obiectului cu contextul anterior nodului curent, al doilea element fiind tagul nodului curent iar al treilea element este orice tag care nu marchează final de propoziție. De asemenea, pentru modul forward, se poate observa că elementele trigramului sunt luate de la stânga la dreapta, specific metodei forward.

Pentru condiția de bigram, logica desfășurării este foarte asemănătoare cu cea descrisă aici 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, după ce s-a obținut nodul cu valoarea maximă, acesta poate fi adăugat în lista de noduri pentru iterația la timpul = i. Acest nod cu valoarea maximă se va calcula pentru fiecare nod calculat anterior dacă există bineînțeles și o legătură de tip bigram între acestea.

Ultimul caz rămas este atunci când algoritmul a ajuns la verificarea unui token care nu mai este primul token din propoziție iar acesta este deja în lista de probabilități de emisie. Acest caz este asemănător cu cel descris anterior pentru cuvintele necunoscute, doar că în loc de calcularea probabilității cuvântului necunoscut cu fiecare tag, se va calcula/prelua probabilitatea de emisie pentru tagurile cuvântului respectiv. Implementarea pentru trigram este următoarea:

**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 tagul actual al cuvântului curent, vn.value fiind probabilitatea nodului anterior. Implementarea bigram este foarte asemănătoare cu cea descrisă anterior, doar că aceastea se aplică doar când modelul a fost ales ca bigram:

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 în lista de noduri actuale, la fel ca în procesul descris pentru verificarea primului token din propoziție, după aceasta, lista este și ea introdusă la rândul ei în matricea grafului Viterbi la timpul = i, în final, ultima linie din matrice fiind sortată descrescător pentru ca primul nod să fie cel cu probabilitatea cea mai mare atunci când este realizat procesul de backtracing.

Funcția de backtracing care se aplică și atât pentru metoda forward cât și pentru metoda backward este următoarea:

**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**);**

**}**

**}**

Această funcție mai întâi va selecta nodul cu valoarea cea mai mare (nodurile sunt sortate la fiecare iterație de token), și va verifica care metodă a fost aleasă (forward/backward). Dacă metoda a fost apelată cu parametrul de forward atunci se va întra în condiția de forward, în această condiție se va salva nodul final cu valoarea maximă într-o listă de istorie și se va face backtrack de la nodul curent spre prevNode, până când nodul anterior nu mai există adică prevNode va fi egal cu null. Toate aceste taguri provenite de la nodurile iterate se vor salva intr-o listă de taguri predicționate. Pentru metoda backward, aceasta doar salvează ultimul nod cu valoarea maximă în lista de istorie, fără să adauge contextul tagului în lista de taguri predicționate pentru fiecare iterație, deoarece ordinea tagurilor rezultate pentru această metodă este diferită față de ordinea de la forward pentru același set de date, salvarea tagurilor fiind un proces eronat.

#### 3.2.4.2 Backward method

Această metodă presupune parcurgerea propoziției/frazei de la dreapta spre stânga, adică de la sfârșit spre început. Această metodă este de cele mai multe ori mult mai bună decât cea forward, ea fiind inversul procesului forward. Totuși, propozițiile/frazele sunt de sine stătătoare, ea poate itera fiecare propoziție din setul de antrenare în ordine (implementarea a fost făcută să înceapă de la finalul setului de test spre început) dar fiecare propoziție va fi evaluată de la final spre început.

Backward method folosește și ea probabilitatea de emisie, probabilitatea cuvintelor necunoscute și probabilitatea de tranziție, doar că la fel ca la forward, primul token/cuvânt de la final nu va putea folosi modelul trigram. Înafara de aceasta, dacă setul de testare este și el evaluat de la final spre început, atunci va trebuii verificat și momentul când se ajunge la primul cuvânt/token din setul de testare fără a evalua si tagurile de început/sfârșit de propoziție, restul codului fiind aproape identic cu cel de la forward cu minimele modificări descrise aici. Începutul metodei de backward, descris în cod, este următorul:

**......**

**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 că iterația începe de la ultimul cuvânt din setul de testare care nu are tagul de sfârșit de propoziție asociat, și continuă până la indexul -1, adică până la începutul de propoziție pentru setul de testare. Indexul -1 este verificat deoarece atunci se poate face backtracing cu metoda backward și la fel și atunci când se ajungle la tagul de sfârșit/început de propoziție (‘.’).

La implementare, nodul anterior va fi acum cel posterior, iar probabilitățile de tranziție vor fi evaluate  de la final spre început:

**......**

**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 anterior este luat din condiția de trigram pentru cuvintele necunoscute, elementele tuplului de trigram sunt acum evaluate de la cel din dreapta spre stânga. La final după ce s-a calculat produsul pentru nodul curent, se verifică dacă este mai mare ca nodul maxim iar dacă este, atunci nodul maxim va deveni nodul curent cu legătura la nodul posterior (în acest caz fiind tot nodul anterior evaluat) la fel ca la metoda de forward. Cu toate că această metodă este foarte asemănătoare cu cea de forward, backtracking-ul pentru a emite tagurile predicționate pentru metoda backward se face doar la final, după ce s-a evaluat fiecare cuvânt/token din setul de testare.

În metoda de backtracking, se salvează o listă de noduri finale cu valoarea maximă, iar când setul de testare a fost iterat complet, atunci se pot emite tagurile finale pentru această metodă:

**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 că această condiție este și ea, la rândul ei, foarte asemănătoare cu metoda de backtracking descrisă aici la **Forward method**, diferența aici fiind că se va itera lista cu istoria nodurilor cu valoarea maximă iar backtrackingul se va face de la nodul curent spre nodul posterior (nextNode) până când acesta devine null (adică s-a evaluat întreaga propoziție/frază).

#### 3.2.4.3 Bidirectional method

Această metodă combină ambele metode menționate anterior, am descris în procesele anterioare că se va păstra o listă cu istoria nodurilor finale pentru ambele metode, având în vedere că fiecare nod din lista cu istoric reprezintă secvența de propoziție/frază evaluată de metoda respectivă, se poate atunci compara fiecare nod din cele 2 liste iar cel cu valoarea cea mai mare va fi ales să îi se facă backtrack pentru a se obține tagurile finale predictionate pentru acea propoziție.

Aceasta presupune că valoarea cea mai mare este cea corectă, deoarece aceasta are o încredere mai mare pentru secvența de taguri acolo unde nodul este cu o probabilitate mai mare. Funcția metodei bidirecționale este următoarea:

**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**);**

**}**

**}**

**}**

Logica algoritmului este la fel cu cel menționat la backtracking pentru metod forward și pentru metoda backward. Pentru cazul când valorile nodurilor sunt egale (foarte rar) atunci cel mai probabil se indică aceleași taguri pentru propoziția/frază respectivă și oricare metodă s-ar alege, rezultatul evaluărilor ar fi exact aceelaș.

În următorul grafic, se poate vedea de câte ori se intră în branch-ul de backward, în cel de forward și de câte ori valorile maxime pentru forward și backward sunt egale:



Figura 3.7 – probabilitatea fiecărei metode să fie alese in funcție de valoarea nodului final

### Evaluation

După ce decodorul a emis tagurile predictionate pentru setul de testare, sistemul trebuie să evalueze tagurile predicționate pentru a vedea câte din acestea au fost predicționate corect. Cum algoritmul acesta este unul de învățare supervizată, există informații legate de tagurile corecte a setului de testare, cu toate că acestea nu sunt folosite în procesul de predicție, ele sunt foarte folositoare atunci când se dorește evaluarea modelului. Evaluarea pentru acest model este împărțită în 2 categorii:

- Evaluarea acuratății generale

- Evaluarea matricii de confuzie

#### 3.2.5.1 General Accuracy

Această metodă simplă de evaluare presupune calcularea numărului de taguri predicționate corect pe (supra) numărul total de taguri predicționate. Aceasta este defapt, acuratețea totală pentru toate cuvintele/token-urile din setul de testare. Pentru acest tip de model, se poate calcula și acuratețea pentru cuvintele cunoscute și separat pentru cuvintele necunoscute, acestea sunt foarte utile deoarece se poate evalua separat acuratețea modelului Markov și cea a modelului pentru cuvintele necunoscute.

Funcția care implementează acuratețea generală este următoarea:

**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**;**

**}**

Această funcție are un parametru de intrare unde se poate seta ce fel de acuratețe se dorește la returnare, ’k+u’ este acuratețea generală, ’k’ este acuratețea pentru cuvintele cunoscute și ’u’ este acuratețea pentru cuvintele necunoscute. Funcția folosește Set-ul de cuvinte necunoscute care a fost încărcat la decodare. Acuratețea finală se va calcula ca wordsHit (incrementat dacă tagul la index-ul i pentru setul de testare este același cu tagul predicționat la index-ul i) supra numărul total de cuvinte. În momentul când se alege acuratețea pentru cuvinte cunoscute, metoda verifică dacă la indexul curent acest cuvânt se află și în setul de cuvinte necunoscute, dacă nu se află atunci acesta continuă cu evaluarea wordsHit, dacă se află atunci se trece la indexul următor fără să fie procesat cuvântul/tokenul curent. Pentru cuvintele necunoscute, procesul este asemănător cu cel al cuvintelor cunoscute descris aici.

#### 3.2.5.2 Confusion matrix

Matricea de confuzie, cunoscută și ca matricea de eroare, este un tabel specific care permite vizualizarea performanței unui algoritm de învățare supervizată. Liniile tabelului reprezintă clasa reală (tagul corect) iar coloanele reprezintă clasa predicționată (tagul predicționat de decodor) [21]. Acest tabel este foarte util atunci când se evaluează algoritmul pe mai multe clase (multi-class classification), se obțin metrici de evaluare pentru fiecare clasă iar metrica totală este media aritmetică a metricilor tuturor claselor (tagurilor).

****

Figura 3.8 – matricea de confuzie – informații [22]

Fiecare clasa unică din setul de testare va avea propria ei matrice de confuzie, cele 4 valori tp, tn, fp, fn vor fi la început inițializate cu 0 și vor crește în timp ce se iterează setul de testare cu setul de clase predicționate. Metricile de evaluare care se pot realiza în urma utilizării unei matricii de confuzie și care sunt implementate în sistem sunt:

**Accuracy**: aceasta ia în calcul aici și cazurile când clasa nu apare nici pentru setul de testare și nici nu a fost predicționată (true negative)

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

**Precision**: este procentajul rezultatelor care sunt relevante. Aceasta încearcă să răspundă la întrebarea: “Ce proporție identificată pozitiv (tp) este și corectă?”

|  |  |
| --- | --- |
|  | (3.18) |

**Recall**: se referă la procentajul tuturor rezultatelor relevante corect clasificate de algoritm. Aceasta încearcă să răspundă la întrebarea: “Ce proporție identificată pozitiv (tp) a fost identificată corect?”

|  |  |
| --- | --- |
|  | (3.19) |

**Specificity (True Negative Rate)**: reprezintă proporția rezultatelor negative, corect identificate ca negative (clasa nu a fost predicționată și nici nu trebuia predicționată).

|  |  |
| --- | --- |
|  | (3.20) |

**F-measure**: reprezintă media armonică dintre precizie și recall.

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

Deobicei, parametrul are valoarea 1, atunci noua formulă se va numi **F1-Score**, ea fiind redusă la forma:

|  |  |
| --- | --- |
|  | (3.22) |

Aceste metrici de evaluare sunt implementate in funcția următoare:

**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 declară un set ClassTags pentru a se putea păstra fiecare clasă unică (tag) din setul de testare. În secțiunea sublinitată, se poate remarca pentru ce condiții cresc cele 4 valori, true negative crește atunci când nici setul de test și tagul predicționat nu sunt aceleași cu tagul curent din iterația setului ClassTags, true positive crește când și setul de test și tagul predicționat sunt egale cu tagul curent, false negative crește doar când setul de test este același cu tagul actual iar false positive crește doar când clasa predicționată este aceași cu tagul curent.

După ce au fost implementați acești algoritmi, rezultatele pot fi afișate pe ecran pentru a putea fi vizualizată performanța sistemului și acuratețea acestuia.

# IV. Rezultate

TAG ACCURACY PRECISION RECALL(TPR) F1-SCORE SPECIFICITY(TNR)

NN 0.9786396 0.956988 0.9574453 0.9572166 0.9856884

OT 0.9985417 0.9918948 0.991861 0.991878 0.9992006

CC 0.99369 0.9188147 0.9750162 0.9460816 0.9948141

JJ 0.9882605 0.8971757 0.9316306 0.9140785 0.992329

PP 0.9940661 0.9773701 0.9759369 0.976653 0.9967076

AT/DT 0.9936931 0.981871 0.9693777 0.9755844 0.9973259

VB 0.9863712 0.9666563 0.9477145 0.9570917 0.9937555

PN 0.9984745 0.9896245 0.9871902 0.9884058 0.9992702

RB 0.9886977 0.9018956 0.8847122 0.8932213 0.9945676

TOTAL 0.9911593 0.9535878 0.957876 0.955579 0.994851

Accuracy for known words: 0.966862

Accuracy for unknown words: 0.8051774

Accuracy on both: 0.9602172

# V. Bibliografie

[1] Dan Jurafsky , Speech and Language Processing, carte online, accesat în data de (29/05/2020): <https://web.stanford.edu/~jurafsky/slp3/8.pdf>

[2] TnT - A statistical POST (2000), articol științific online, accesat în data de (29/05/2020): <http://www.coli.uni-saarland.de/~thorsten/publications/Brants-ANLP00.pdf>

[3] Viterbi Algorithm wikipedia, accesat în data de (29/05/2020): <https://en.wikipedia.org/wiki/Viterbi_algorithm>

[4] Part of speech Tagging, curs 4 (2013), accesat în data de (29/05/2020): <http://ivan-titov.org/teaching/nlmi-15/lecture-4.pdf>

[5] Part of speech Tagging, curs 5 (2013), accesat în data de (29/05/2020): <https://staff.fnwi.uva.nl/k.simaan/D-Courses2013/D-NLMI2013/college5.pdf>

[6] Proiect POST open-source pe github, accesat în data de (29/05/2020): <https://github.com/ST4NSB/part-of-speech-tagging>

[7] Brown Corpus manual, informații legate de setul de date, accesat în data de (29/05/2020): <http://korpus.uib.no/icame/manuals/BROWN/INDEX.HTM>

[8] Cross-Validation - wikipedia, accesat în data de e (29/05/2020): <https://en.wikipedia.org/wiki/Cross-validation_(statistics)>

[9] Quantinsi Cross-validation, articol online, accesat în data de (29/05/2020): <https://blog.quantinsti.com/cross-validation-machine-learning-trading-models/>

[10] Fisher-Yates Shuffling algoritm - wikipedia, accesat în data de (29/05/2020): <https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle>

[11] Părți de vorbire în gramatica limbii engleze (1), accesat în data de (29/05/2020): <https://www.grammar.cl/english/parts-of-speech.htm>

[12] Părți de vorbire în gramatica limbii engleze (2), accesat în data de (29/05/2020): <http://www.butte.edu/departments/cas/tipsheets/grammar/parts_of_speech.html>

[13] Părți de vorbire în gramatica limbii engleze (3), accesat în data de (29/05/2020): <https://www.englishclub.com/grammar/parts-of-speech.htm>

[14] Părți de vorbire în gramatica limbii engleze (4), accesat în data de (29/05/2020): <https://www.english-grammar-revolution.com/parts-of-speech.html>

[15] LINQ microsoft docs, documentație online, accesat în data de (29/05/2020): <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>

[16] POS Implications of Affixes (1966), articol științific online, accesat în data de (06/06/2020): <https://pdfs.semanticscholar.org/7008/6ddca220c59a215e815da69205bca2022158.pdf>

[17] Grammar & Structure : prefixes & suffixes, articol online, accesat în data de (06/06/2020): <https://web2.uvcs.uvic.ca/courses/elc/sample/beginner/gs/gs_55_1.htm>

[18] ESL Library: Suffixes that show the POS (2016), articol online, accesat în data de (06/06/2020): <https://esllibrary.com/blog/english-word-endings-suffixes-that-show-the-part-of-speech/>

[19] Additive smoothing – wikipedia, accesat în data de (06/06/2020): <https://en.wikipedia.org/wiki/Additive_smoothing>

[20] Task Class in .NET, documentație online, accesat în data de (06/06/2020): <https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=netcore-3.1>

[21] Confusion matrix – wikipedia, accesat în data de (06/06/2020): <https://en.wikipedia.org/wiki/Confusion_matrix>

[22] Daniel Morariu & Radu Crețulescu – Text mining, tehnici de clasificare și clustering al documentelor (2012)