**Coperta**

**Cuprins**

**I. Prezentare temă**

**II. Algoritmi de învăţare**

Ecuatia cu decoder produs tranzitie \* emisie

**III. Consideraţii practice**

* 1. **Descriere generală proiect**

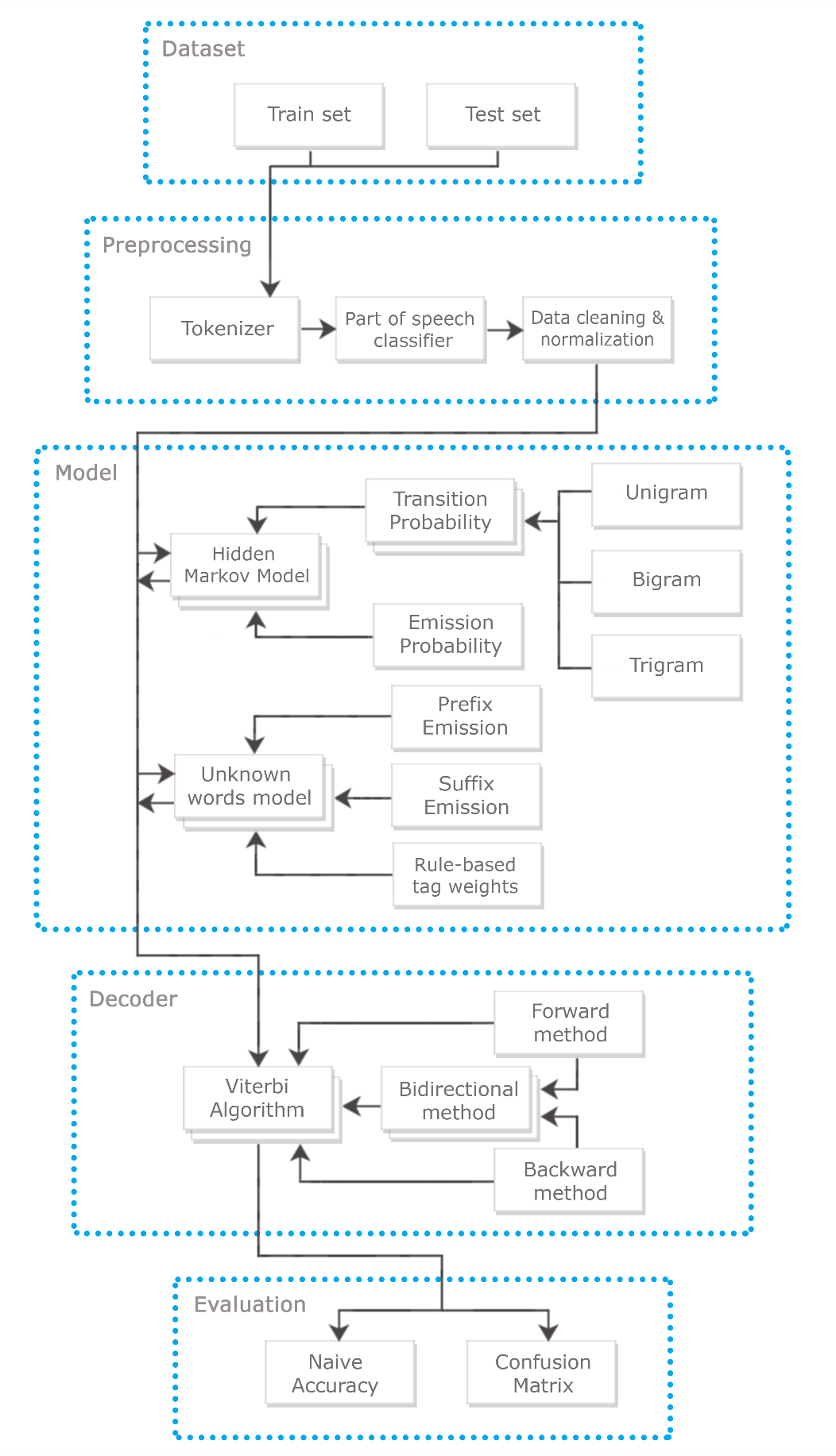


Figura 3.1 - Arhitectura proiectului

* 1. **Descriere separată bloc**

**3.2.1 Dataset**

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

A. Presă: Reportaje – 44 documente

B. Presă: Editorial – 27 documente

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

D. Religie – 17 documente

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

F. Folclor popular – 48 documente

G. Scrisori, bibliografii, biografii – 75 documente

H. Diverse – 30 documente

J. Articole stiintifice– 80 documente

TOTAL – 374 documente

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

K. Fictiune generala – 29 documente

L. Mister și ficțiune detectiva – 24 documente

M. Opere științifico-fantastice – 6 documente

N. Aventura și ficțiunea western – 29 documente

P. Povesti romantice și de dragoste – 29 documente

R. Umor – 9 documente

TOTAL – 126 documente

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



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

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

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

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

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

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

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

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

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

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

**3.2.1.1 Train & Test Set**

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

**a) Impartire 70% - setul de antrenament, 30% - setul de testare**

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

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

**b) Cross-Validation**

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

**K folds cross-validation**

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



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

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

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

k - numarul de fold-uri alese

– acuratetea pe fold-ul i

**Implementare K folds cross-validation**

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

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

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

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

**{**

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

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

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

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

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

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

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

**{**

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

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

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

**{**

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

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

**{**

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

**}**

**else**

**{**

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

**}**

**}**

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

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

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

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

**}**

**}**

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

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

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

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

Pentru obtinerea textului din foldere in lista avem urmatoarea implementare:

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

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

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

**{**

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

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

**}**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Random rng **=** **new** Random**();** // random seed

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**;**

* + 1. **Preprocessing**

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

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

**3.2.2.1 Sentence tokenization**

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

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

ex. I’m home but he is not home.

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

**Implementare sentence tokenization**

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

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

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

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

**{**

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

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

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

**{**

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

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

**}**

**}**

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

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

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

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

**public** struct WordTag

**{**

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

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

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

**{**

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

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

**}**

**}**

, iar codul de despartire si adaugare in lista:

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

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

**{**

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

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

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

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

**{**

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

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

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

**}**

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

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

**}**

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

**3.2.2.2 Part of speech classifier**

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

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

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

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

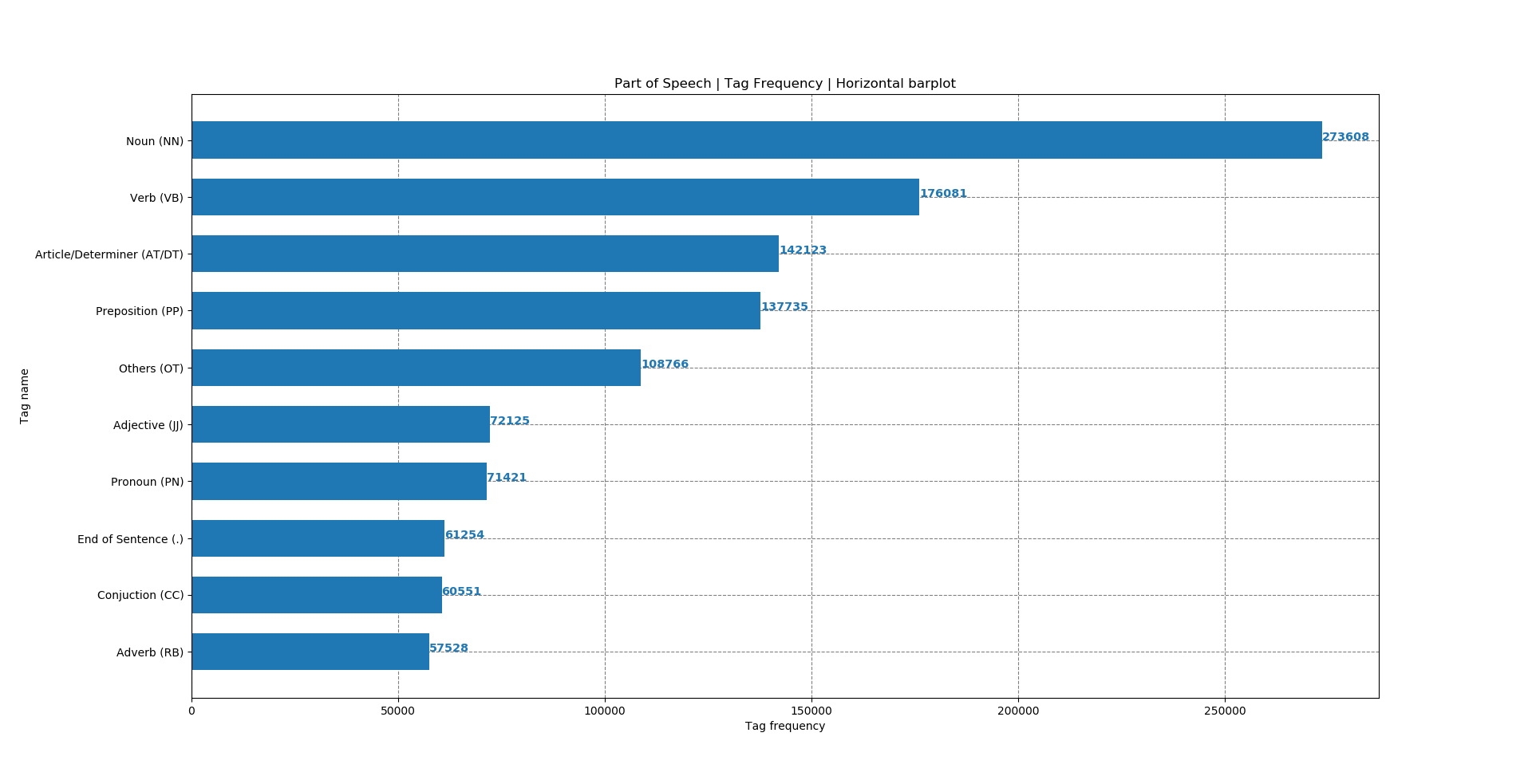


Figura 3.4 – Frecvenţa noilor tag-uri



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

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

**Implementare part of speech classifier**

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

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

**{**

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

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

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

**}**

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

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

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

**{**

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

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

**{**

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

**{**

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

**return** tagIndex**;**

**}**

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

**{**

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

**return** tagIndex**;**

**}**

**}**

**}**

**return** tagIndex**;**

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

Lista BrownCorpusTags contine urmatoarele tag-uri:

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

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

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

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

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

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

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

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

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

"."

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

Conditia de clasificare ConvertBrownTagToHierarchicTag:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**else**

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

**return** tag**;**

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

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

GetTagIndexForConversion va returna 6,

ConvertBrownTagToHierarchicTag va returna “NN”

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

**3.2.2.3 Data cleaning & normalization**

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

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

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

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

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

**Implementare Data cleaning & normalization**

Implementare Preprocessing pipeline:

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

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

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

**{**

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

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

**{**

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

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

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

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

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

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

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

**continue;**

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

**}**

**return** newWords**;**

**}**

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

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

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

**)**

**{**

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

**}**

Implementare functie de eliminare cifre:

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

**{**

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

**return** word**;**

**else**

**{**

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

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

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

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

**return** output**;**

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

**}**

**}**

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

* + 1. **Model**

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

* Hidden Markov model
* Unknown words model
  + - 1. **Hidden Markov Model**

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

1. **Emission probability**

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

|  |  |
| --- | --- |
|  | (3.2) |
| Aceasta probabilitate este calculata pentru setul de testare dar numaratorile sunt facute in setul de antrenare. | | |
| ex. … ***he/???***went/VB home/NN … | (3.3) |

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

**Implementare emission probability**

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

**public** class EmissionModel

**{**

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

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

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

**{**

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

**}**

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

**{**

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

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

**}**

**}**

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

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

**this.**WordCapitalizedTagsEmissionFrequence **=**

**new** List**<**EmissionModel**>();**

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

**{**

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

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

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

**{**

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

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

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

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

**}**

**else**

**{**

var tag **=** wmFind**.**TagFreq**.**

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

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

**{**

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

**}**

**else**

**{**

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

**}**

**}**

**}**

Se aplica LINQ (Linq docs, 2017), pentru a cauta daca in lista cu frecvente avem deja cuvantul pe care il iteram, daca nu gasim cuvantul in lista atunci creem un obiect de tip EmissionModel si adaugam cuvantul si tag-ul aferent acestuia cu un count = 1, dupa care il adaugam in lista WordCapitalizedTagsEmissionFrequence. Daca totusi gasim cuvantul in lista, atunci cautam tag-ul aferent acestuia, daca nu gasim tag-ul atunci adaugam noul tag la dictionar, altfel daca acel tag pe care il cautam exista, atunci incrementam count-ul tag-ului de la cheia unde l-am gasit. Aceeasi procedura si pentru cuvintele cu litera mica:

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

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

**{**

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

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

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

**{**

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

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

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

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

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

**}**

**else**

**{**

var tag **=** wmFind**.**TagFreq**.**

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

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

**{**

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

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

**}**

**else**

**{**

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

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

**}**

**}**

**}**

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

**public** class EmissionProbabilisticModel

**{**

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

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

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

**{**

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

**}**

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

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

**{**

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

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

**}**

**}**

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

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

**{**

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

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

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

EmissionProbabilisticModel wFind = WordTagsEmissionProbabilities**.**

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

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

**{**

EmissionProbabilisticModel epModel **=**

**new** EmissionProbabilisticModel**();**

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

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

**{**

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

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

double pwiti **=** **(**double**)**tf**.**Value **/** cti**;** // Em. prob

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

**}**

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

**}**

**}**

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

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

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

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

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

**Transition probability**

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

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

|  |  |
| --- | --- |
| ex. … Anna/NN ***likes/VB???*** ice-cream/JJ … | (3.5) |

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

**Bigram & unigram**

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

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

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

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

**Implementare Bigram & unigram**

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

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

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

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

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

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

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

**{**

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

**{**

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

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

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

**continue;**

**}**

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

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

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

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

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

**{**

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

**}**

**else**

**{**

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

**}**

**}**

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

Pentru unigram, implementarea este urmatoarea:

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

**{**

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

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

**{**

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

**}**

**else**

**{**

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

**}**

**}**

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

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

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

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

x – numarul tuturor partilor de vorbire individuale din corpus

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

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

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

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

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

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

**{**

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

**{**

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

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

**}**

**}**

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

**{**

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

**{**

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

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

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

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

**}**

**}**

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

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

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

**Trigram**

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

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

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

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

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

**Implementare trigram**

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

Contorizari aparitii:

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

Probabilitati:

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

Algoritmul implementat de numarare a frecventelor de aparitie:

**this.**TrigramTransitionFrequence **=**

**new** Dictionary**<**Tuple**<**string**,** string**,** string**>,** int**>();**

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

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

**{**

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

**{**

**this.**TrigramTransitionFrequence**.**

Add**(new** Tuple**<**string**,** string**,** string**>(**

"."**,**

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

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

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

**continue;**

**}**

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

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

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

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

**continue;**

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

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

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

**{**

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

**}**

**else**

**{**

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

**}**

**}**

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

Functia de calcul a probabilitatii trigram-ului:

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

**{**

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

**{**

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

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

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

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

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

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

**}**

**}**

**Trigram & bigram smoothing**

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

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

Pentru valorile sunt estimate prin interpolarea eliminata. Pseudocodul general este descris:



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

Se poate observa cum probabilitatile fiecarui n-gram sunt scazute cu 1 si la numarator si la numitor, asta inseamna ca algoritmul ia in calcul date care nu au aparut in setul de antrenare. Cu toate acestea mentionate, interpolarea liniara nu este folositoare numai pentru a trata tupluri lipsa dar si pentru a seta ponderi fiecarui n-gram separat. Folosind aceasta functie, acuratetea predictorului creste, in cazul meu, cresterea este mica deoarece setul de antrenare este clasificat in 10 clase (tag-uri) reprezentative si de aceea nu reuseste sa gaseasca date lipsa, dar foloseste ponderile pentru a estima probabilitatea finala in functie de fiecare n-gram.

Inafara de interpolarea facuta pentru trigram, s-a realizat si o interpolare facuta pe bigram, aceasta este prezentata mai departe la implemetare.

**Implementare deleted interpolation**

+ nu uita sa prezinti tot algoritmu trigram+bigram+task

* + 1. **Decoder**
    2. **Evaluation**

**IV. Rezultate**

**V. Anexa 1: Exemplu etichetare parte de vorbire**

**VI. Bibliografie**

[1] Jurafsky slp3-post (2019): <https://web.stanford.edu/~jurafsky/slp3/8.pdf>

[2] Brants-ANL (2000): <http://www.coli.uni-saarland.de/~thorsten/publications/Brants-ANLP00.pdf>

Proiectul open source: <https://github.com/ST4NSB/part-of-speech-tagging>

[3] Brown Corpus manual: <http://korpus.uib.no/icame/manuals/BROWN/INDEX.HTM>

Linq docs (2017): <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>

Brown Corpus wiki: <https://en.wikipedia.org/wiki/Brown_Corpus>

Brown Corpus data-set download: <https://archive.org/details/BrownCorpus>

Cross-Validation informatii: <https://en.wikipedia.org/wiki/Cross-validation_(statistics)>

Quantinsi Cross-validation exemple: <https://blog.quantinsti.com/cross-validation-machine-learning-trading-models/>

Cross-Validation exemple implementari: <https://scikit-learn.org/stable/modules/cross_validation.html>

Fisher-Yates Shuffling algoritm: <https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle>

Fisher-Yates Shuffling solutions (2009): <https://stackoverflow.com/questions/273313/randomize-a-listt>

File Reader in c# informatii: <https://docs.microsoft.com/en-us/dotnet/standard/io/how-to-read-text-from-a-file>

Tokenization (Analiza lexicala) wiki: <https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization>

Parti de vorbire in gramatica limbii engleze (1): <https://www.grammar.cl/english/parts-of-speech.htm>

Parti de vorbire in gramatica limbii engleze (2): <http://www.butte.edu/departments/cas/tipsheets/grammar/parts_of_speech.html>

Parti de vorbire in gramatica limbii engleze (3): <https://www.englishclub.com/grammar/parts-of-speech.htm>

Parti de vorbire in gramatica limbii engleze (4): <https://www.english-grammar-revolution.com/parts-of-speech.html>