UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE INGINERIE

DEPARTAMENTUL DE CALCULATOARE ŞI INGINERIE ELECTRICĂ

PROIECT DE DIPLOMĂ

Îndrumător: Conf. dr. ing. Morariu Daniel

Absolvent: Bărbulescu Adrian

Specializarea: Ingineria Sistemelor Multimedia

- Sibiu, 2020 -

UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE INGINERIE

DEPARTAMENTUL DE CALCULATOARE ŞI INGINERIE ELECTRICĂ

Etichetarea părților de vorbire

Îndrumător: Conf. dr. ing. Morariu Daniel

Absolvent: Bărbulescu Adrian

Specializarea: Ingineria Sistemelor Multimedia

Cuprins

I Prezentare temă

Partea de vorbire (în engleză *part of speech*), este o clasă de cuvinte stabilită după sensul lor lexical și după caracteristicile lor morfologice și sintactice. Acestea au fost recunoscute în lingvistică de mult timp, gramaticianul grec *Dionysius Thrax* (secolul I î.Hr) a clasat cuvintele în opt părți de vorbire: substantiv, verb, participiu, articol (incluzând și pronumele relativ), pronume, prepoziție, adverb și conjuncție [1], [2]. Din toate acestea menționate, părțile de vorbire folosite și în ziua de azi, care se învață încă din școala primară sunt: substantivul, verbul, adjectivul, adverbul, prepoziția, conjuncția, pronumele și interjecția.

Etichetarea părților de vorbire (în engleză *part of speech tagging*) este procesul în care cuvintele dintr-o propoziție vor primi o clasă etichetă (în engleză numit și *tag*) cu partea de vorbire a acestora. Etichetarea este un proces de dezambiguizare, cuvintele dintr-o propoziție sunt ambigue, ele pot avea mai multe părți de vorbire în diferite contexte iar scopul etichetării este acela de a alege partea de vorbire (numit și tag) corectă în contextul respectiv.

În această lucrare este prezentat un sistem automat care analizează un text din limba engleză și încearcă să eticheteze corect părțile de vorbire, sistemul folosește diferiți algoritmi de învățare din domeniul învățării automate (în engleză *machine learning*, prescurtat *ML*) și algoritmi din domeniul prelucrării limbajului natural (în engleză *natural language processing*, prescurtat *NLP*). Un astfel de sistem de etichetare este foarte util în diferite aplicații din domeniul prelucrării limbajului natural, de exemplu, un program care se ocupă cu indexarea textelor și cu regăsirea acestora poate folosi un sistem de etichetare pentru a determina tagurile cuvintelor din text, mai apoi salvând cuvintele de interes (keywords) ca index. Acestea pot fi salvate pe baza cunoștințelor că substantivul este, de obicei, subiectul propoziției, acesta descrie cuvinte care sunt nume de persoane, locuri, lucruri, verbul descrie cuvinte care fac referință la acțiuni și procese, adjectivul include termeni care descriu proprietăți, calități și relația cu substantivul, adverbul de obicei exprimă timpul, frecvența, gradul, nivelul de certitudine, iar restul părților de vorbire pot descrie cuvinte de legătură, numere, reacții, cuvinte din altă limbă, etc.

Procesul de etichetare a părților de vorbire este un pas important în procesarea vorbirii și a limbajului, în ziua de azi multe companii se folosesc de algoritmi de procesare a vorbirii și a limbajului pentru a dezvolta diferite aplicații pentru utilizatori. Exemple de aplicații utilizate în viața reală sunt următoarele:

- în E-commerce și IT, în locul unei persoane care are rolul de helpdesk, se folosesc chatboți inteligenți care oferă suport utilizatorilor
- diverse tehnologii de asistență virtuală precum Alexa sau Siri utilizate într-o casă inteligentă pot procesa comenzi vocale,
- site-urile de socializare colectează și procesează datele utilizatorilor oferind reclame sau pagini web pe interesul acestora,
- majoritatea motoarelor de căutare cunoscute precum Google, Bing, DuckDuckGo folosesc tehnici de analiză și procesare a textului pentru a oferi cele mai relevante rezultate utilizatorilor,
- aplicațiile de pe smartphone care folosesc tastatura smartscreen pentru conversații online precum WhatsApp, Facebook Messenger, au incorporat și tehnici de predicție, autocompletare și autocorectare a textului introdus de utilizator,
- diferiți algoritmi pentru a marca și a elimina știrile false despre pandemia de COVID-19 au fost implementate recent pe rețelele de socializare și în motoarele de căutare, etc.

Alegerea acestei teme pentru proiectul de diplomă a fost datorită interesului meu pentru domeniul inteligenței artificiale (prelucrarea limbajului natural fiind un subdomeniu din acest domeniu) și aplicarea algoritmilor *State of the art* de invățare automată și de prelucrare a limbajului natural.

În următoarele capitole, voi prezenta diferiți algoritmi de învățare din domeniul ML & NLP bazate pe modele statistice precum *lanțuri Markov* și *modele Markov cu stări ascunse* dar și modele bazate pe reguli pentru a determina partea de vorbire a unui cuvânt, metode de decodificare a unei secvențe Markov, metode de curățare și de normalizare a setului de date, diferite metrici de evaluare precum acuratețea, precizia, recall-ul, specificitatea, f-measure-ul și rezultatele obținute pentru sistemul de etichetare folosit.

II Considerații teoretice

În acest capitol se vor prezenta bazele teoretice pentru algoritmii folosiți în sistemul de etichetare a părților de vorbire implementat în această lucrare. Pe lângă acestea, se vor prezenta și informații despre domeniile aferente acestei teme (inteligența artificială, prelucrarea limbajului natural), tipuri de sisteme de etichetare și tehnici de evaluare a modelelor propuse.

2.1 Inteligența artificială (AI)

În termeni simpli, inteligența artificială (IA) se referă la sisteme sau la mașini care imită inteligența umană, pentru a efectua diverse activități și care se pot îmbunătăți iterativ pe baza informațiilor pe care le colectează [3]. Kaplan și Haenlein definesc IA ca fiind „capacitatea unui sistem de a interpreta corect datele externe, de a învăța din astfel de date și de a folosi ceea ce a învățat pentru a-și atinge obiectivele și sarcinile specifice printr-o adaptare flexibilă". Termenul „inteligență artificială" este utilizat colocvial pentru a descrie mașinile care imită funcțiile „cognitive" pe care le asociază oamenii cu alte minți umane, cum ar fi „învățarea" și „rezolvarea problemelor" [4].

Domeniul a fost întemeiat pe afirmația că inteligența umană „poate fi descrisă atât de precis încât poate fi făcută o mașină pentru a o simula" [4]. Acest lucru ridică argumente filosofice cu privire la natura minții și la etica creării de ființe artificiale dotate cu inteligență umană, care sunt chestiuni care au fost explorate încă din Antichitate .Acest domeniu a fost întemeiat la Colegiul Dathmouth în 1956, Allen Newell, Herbert Simon, John McCarthy, Marvin Minksy și Arthur Samuel au devenit fondatorii și liderii cercetării IA. La sfârșitul anilor 1990 și începutul secolului al XXI-lea, inteligența artificială a început să fie folosită pentru logistică, data mining, diagnostic medical și în alte domenii. Succesul s-a datorat creșterii puterii de calcul (Legea lui Moore), accentului mai mare pus pe rezolvarea problemelor specifice, legături noi între IA și alte domenii (cum ar fi statistica, economia și matematica) și angajamentul cercetătorilor față de metodele matematice și standardele științifice [4]. Exemple de utilizare a inteligenței artificiale sunt: traducerile automate, roboți, programe care joacă jocuri pe tablă ca dame, go (sistemul de joc AlphaGo, creat de Google Deep Mind a reușit să îl învingă, în martie 2016, pe Lee Sedol, considerat cel mai bun jucător de go), backgammon, șah (Deep Blue a fost primul sistem de joc care l-a învins pe campionul mondial, Garry Kasparov, în mai 1997), diagnoză medicală, planificarea automată, recunoașterea scrisului, etichetarea părților de vorbire, etc.

Inteligența artificială se folosește de multe domenii precum: filozofia, matematica, economia, ingineria calculatoarelor, lingvistica, psihologia, etc. și are ca scop crearea sistemelor inteligente pentru subdomenile ei: învățarea automată (machine learning) și aprofundată (deep learning), viziunea calculatoarelor (computer vision), prelucrarea limbajului natural (natural language processing), etc. Chiar dacă aceste tehnici au condus, sau nu, la o mai bună înțelegere a minții, este evident că aceste dezvoltări vor conduce la o nouă tehnologie, una inteligentă care poate avea efecte dramatice în societate. Sisteme IA experimentale au generat deja, interes și entuziasm, în industrie și au fost dezvoltate comercial [5].

2.2 Prelucrarea limbajului natural (NLP)

Prelucrarea limbajului natural, în engleză natural language processing prescurtat NLP, este un subdomeniu al lingvisticii, științei calculatoarelor și inteligenței artificiale, care se ocupă cu interacțiunea dintre calculatoare și limbajul uman [6]. Obiectivul principal al acestui domeniu este ca un calculator să poată citi, descifra și înțelege limbajul uman într-o manieră avansată. Prelucrarea limbajului natural încearcă, în principal, să rezolve probleme și să vină cu algoritmi în

recunoașterea vorbirii, înțelegerea și generarea limbajului uman. Multe tehnici NLP se bazează pe algoritmi de învățare automată pentru a dobândi sens din limbajul uman.

Aplicații care se folosesc de domeniul de prelucrare a limbajului natural sunt: Google translate pentru a traduce cuvinte în altă limbă, Microsoft Word care folosește un sistem de verificare a corectitudinii documentelor din punct de vedere gramatic, aplicații pe smartphone care pot recunoaște vocea posesorului, asistenți personali virtuali ca OK Google, Cortana, sistem care etichetează automat e-mail-urile de tip spam, aplicații ce returnează rezumatul unui text sau document, etc.

Istoria prelucrării limbajului natural a început prin anii 1950-1951, cu toate că lucrări din acest domeniu se pot găsi și în perioadele anterioare. În 1950, Alan Turing a publicat un articol numit „Computing Machinery and Intelligence" care a propus un concept numit *Testul Turing* care este acum un criteriu de testare a inteligenței unui calculator [6]. Testul Turing presupune un experiment în care se testează dacă modul de gândire al calculatoarelor poate fi asemănător cu cel al oamenilor. Unui calculator și unei persoane le sunt puse niște întrebări de o altă persoană de tip tester, acesta neștiind dacă adresează întrebări unui calculator sau unei persoane, testul Turing este trecut atunci când mașina oferă răspunsuri similare ca cele a unei persoane, testerul neputând să determine dacă adresează întrebări unui calculator sau unei persoane.

Până în anul 1980, majoritatea sistemelor de prelucrare a limbajului natural erau bazate pe reguli complexe scrise manual. Începând din 1980, tehniciile NLP au început să folosească algoritmi din domeniul învățării automate. Unul din cei mai utilizați algoritmi la vremea aceea, arborele de decizie, a produs sisteme bazate pe reguli similare cu cele bazate pe reguli scrise manual. Etichetarea părții de vorbire a introdus modelul Markov cu stări ascunse (în engleză Hidden Markov Model, prescurtat HMM), domeniului de prelucrare a limbajului natural iar de atunci, studiul modelelor statistice a crescut considerabil.

Din 2010, metodele de învățare bazate pe reprezentare (representation learning) și rețelele neuronale aprofundate (deep neural networks) au ajuns să fie foarte utilizate în acest domeniu de prelucrare a limbajului natural deoarece acestea au reușit să atingă rezultate foarte bune. Tehnici populare care includ folosirea algoritmilor de încorporare a cuvintelor (word embeddings) pentru a obține propietățile semantice ale cuvintelor și pentru a crește procesul de învățare la un nivel mult mai avansat (de exemplu pentru procesul de a răspunde la întrebări), în loc de a se baza pe un pipeline de sarcini intermediare separate, de exemplu un pipeline care are o sarcină de etichetare a părților de vorbire și una de executare a procesului de parsare a dependențelor (dependency parsing) și de creare a arborelui parsat (parse tree).

În domeniul de prelucrare a limbajului natural, există mai multe tehnici utilizate în aplicații, unele din acestea reprezintă direct o aplicație întreagă folosită în industrie, alte tehnici NLP sunt subsarcini care sunt folosite pentru a rezolva o problemă mai complexă. Aceste tehnici sunt împărțite în următoarele categorii:

- **Syntax**: lemmatization (returnarea formei de dicționar a unui cuvânt), etichetarea părților de vorbire, parsarea (creearea unui arbore parsat), stemming (returnarea formei de rădăcină a unui cuvânt)
- **Semantics**: traducerea mașină (traducerea unui text dintr-o limbă în alta), generarea limbajului (convertirea informațiilor din baza de date a calculatorului într-o limbă vorbită de oameni), răspunsuri la întrebări (crearea unui soft care recunoaște contextul întrebării și reușește să returneze un răspuns valid)

- **Discourse**: reducerea unui text la forma de rezumat automată, analiza unui discurs și evidențierea cuvintelor importante
- **Speech**: recunoașterea vorbirii, convertirea de la text la voce, segmentarea vorbirii (separarea cuvintelor folosite într-o înregistrare de voce)
- **Dialogue**: conceperea unui sistem care poate ține un dialog cu un utilizator
- **Cognition**: crearea unui sistem care poate să dobândească cunoștințe prin gândire, experiențe și simțuri.

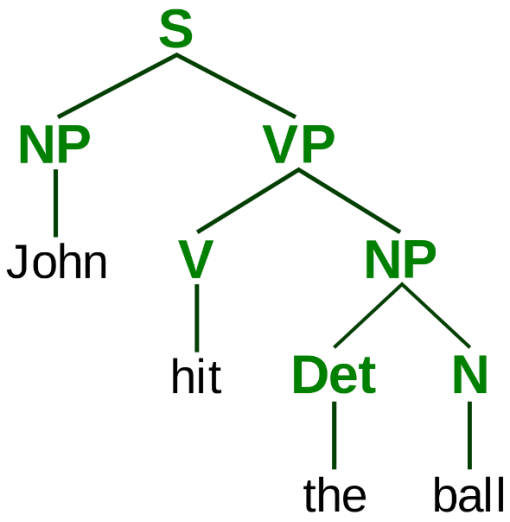

Figura 2.1 – un arbore parsat [4]

Prelucrarea limbajului natural joacă un rol important în susținerea relației de interacțiune dintre om și calculator. Se așteaptă ca în viitor, mașinile de calcul să ajungă la un nivel mult mai bun de recunoaștere și înțelegere a limbilor vorbite de oameni [7].

2.3 Etichetarea părților de vorbire

O aplicație a prelucrării limbajului natural este etichetarea părților de vorbire. Aceasta presupune conceperea unui sistem care primește la intrare un text într-o limbă oarecare și returnează partea de vorbire fiecărui token (cuvânt) din text. Sistemul prezentat în această lucrare va lucra doar pe texte și pe exemple din limba engleză. Părțile de vorbire principale din limba engleză sunt:

- substantiv (noun): his green **eyes** are beautiful ; **Michael** went to the shop
- verb (verb): he **went** to the library ; she **eats** very quickly
- pronume (pronoun): **we** are here ; **They** may ask first
- adjectiv (adjective): she lives in a **beautiful** house ; Jim is a **part-time** worker
- adverb (adverb): I worked **hard** ; The party went **badly**
- prepoziție (preposition): They sat **on** the chair ; There is some food left **in** the fridge
- conjuncție (conjunction): I tried it **but** I don't like it ; I'd like a bike **for** commuting to work
- articol (article): this is **the** place
- determinant (determiner): **A** dog is **a** good pet ; he didn't like **that** car
- numere cardinale (cardinal numbers): **Two** cards are enoughs ; he is **16** years old
- interjecție (interjection): **Ahh**, that feels wonderful ; **Well**, **duh**! That was not very smart

Procesul de etichetare este un proces de dezambiguizare, cuvintele pot avea mai multe părți de vorbire în diferite propoziții, scopul unui sistem de etichetare fiind acela de a selecta tagul potrivit pentru propoziția respectivă. De exemplu, în propoziția „Book that flight", cuvântul Book aici are tagul de verb dar în propoziția „Give me the book!!", cuvântul book este substantiv. Cu toate că majoritatea cuvintelor (aproximativ 85-86%) nu sunt ambigue (numele unei persoane va fi mereu substantiv), restul de 14-15% sunt cuvinte uzuale întâlnite des într-un text (de exemplu Brown corpus sau WSJ corpus) iar de aceea, aproximativ 55-67% de cuvinte din text sunt ambigue [8]. Cele mai frecvente cuvinte ambigue întâlnite în text sunt: *that, back, down, put* și *set*.

Există mai multe tipuri de algoritmi folosiți pentru a obține tagurile într-un sistem de etichetare a părților de vorbire:

- Metode bazate pe programarea dinamică - Steven DeRose și Ken Church au dezvoltat în 1987 diferiți algoritmi pentru programarea dinamică care să obțină tagurile dintr-un text. Metodele dezvoltate de aceștia sunt similare algoritmului Viterbi cunoscut de ceva timp și în alte domenii. DeRose folosea o tabelă de perechi, iar Church folosea o tabelă de tripleți și o metodă de estimare a valorilor pentru tripleții rari sau nonexistenți în Brown Corpus. Ambele metode au reușit să obțină o acuratețe de peste 95%. [10].
- Metode bazate pe modelul Markov cu stări ascunse – algoritmi pentru învățarea supervizată, modelul Markov va fi descris ulterior în detaliu
- Metode bazate pe învățare nesupervizată – aceste tehnici folosesc un set de antrenament care nu a fost etichetat cu tagurile cuvintelor și încearcă să găsească setul de taguri prin inducție. Acestea observă forma cuvântului și derivă categoriile părților de vorbire [10].
- Metode bazate pe învățarea automată precum support vector machine, maximum entropy classifier, neural network perceptron, nearest-neightbor, toate acestea reușesc să ajungă la o acuratețe de peste 95% [10].

Majoritatea sistemelor de etichetare a părților de vorbire fac parte din 3 categorii importante. Acestea sunt:

1. bazate pe reguli concrete de etichetare (Rule-based POS Tagging)
2. bazate pe procese stohastice (Stochastic POS Tagging)
3. bazate pe transformări (Transformation-based Tagging)

În această lucrare se vor prioritiza algoritmii de etichetare de tip stohastic precum modelul Markov cu stări ascunse și metodele de decodificare ale acestuia.

2.3.1 Sistem de etichetare bazat pe reguli concrete de etichetare

Una dintre cele mai vechi tehnici de etichetare este etichetarea bazată pe reguli. Sistemele bazate pe reguli folosesc un dicționar pentru tagurile posibile fiecărui cuvânt. Dacă cuvântul are mai multe taguri posibile, atunci acesta folosește niște reguli scrise manual pentru a identifica tagul corect. Deambiguizarea poate fi efectuată și aici, analizând caracteristicile lingvistice ale unui cuvânt, împreună cu tokenul (cuvântul) precedent, precum și tokenul următor. De exemplu, se presupune că dacă cuvântul precedent al unui cuvânt este articol, atunci cuvântul trebuie să fie un substantiv [9].

Cum sugerează și numele, toate aceste informații într-un sistem de etichetare bazat pe reguli sunt codate sub formă de reguli, acestea pot fi:

- reguli bazate pe context

- expresii obișnuite compilate sub forma unui automat cu stări finite

De asemenea, sistemele de etichetare bazate pe reguli pot avea 2 etape de predicție a tagului:

- Prima etapă: acesta folosește un dicționar pentru a atribuii fiecărui cuvânt o listă de taguri potențiale
- A doua etapă: acesta se va folosi de regulile scrise manual pentru a emite doar un singur tag din lista de taguri construită în prima etapă

Proprietăți ale sistemelor de etichetare bazate pe reguli:

- acestea sunt sisteme de etichetare bazate pe cunoștințe apriori
- de cele mai multe ori, regulile în aceste sisteme sunt scrise manual și nu sunt deduse de algoritmi
- informația este scrisă sub formă de reguli
- există un număr limitat de reguli
- netezirea și modelarea limbajului este definită explicit în sistem prin utlizarea regulilor

Taggerul RDRPOSTagger [11] folosește un sistem de etichetare bazat pe reguli de desfacere.

2.3.2 Sistem de etichetare bazat pe procese stohastice

O altă tehnică de a eticheta părțile de vorbire este cea bazată pe procese stohastice. Aici intervine întrebarea „Ce tip de model reprezintă unul stohastic?”. Modelul care include probabilități (statistice) și frecvențe de apariție pentru taguri & cuvinte poate fi numit un model de tip stohastic. Orice altă abordare diferită a problemei poate fi considerată un proces stohastic [9].

Cele mai simple abordări pentru a crea un sistem bazat pe procese stohastice sunt:

- Frecvența de apariție a cuvintelor – această abordare colectează frecvențele de apariție a fiecărui cuvânt cu fiecare tag asociat, tagul final fiind ales pe baza valorii maxime în tabelul de frecvențe.
- Probabilitatea secvențelor de taguri – o altă abordare este de a colecta o listă de probabilități cu secvențele tagurilor apărute în setul de date. Această tehnică mai este numită și abordarea n-gram deoarece cel mai potrivit tag pentru un cuvânt este determinat de probabilitatea dată de cele n taguri precedente.

Proprietăți ale sistemelor de etichetare bazate pe procese stohastice:

- Acest sistem este bazat pe probabilitatea de apariție a tagului
- Necesită un set de antrenament pentru a colecta frecvențele de apariție
- Cuvintele care nu apar în setul de antrenament nu vor avea probabilități (sau probabilitate egală cu 0)
- Folosește un set de testare diferit de setul de antrenament
- Este cel mai simplu sistem deoarece alege cele mai frecvente taguri asociate la cuvintele din setul de testare

Taggerul POS Stanford [12] folosește un sistem de etichetare bazat pe procese stohastice (cu modelul Maximum-entropy Markov).

2.3.3 Sistem de etichetare bazat pe transformări

Numit și Brill tagging [13], acesta este o instanță a învățării bazate pe transformare (transformation-based learning prescurtat și TBL) care este de fapt un algoritm bazat pe reguli

pentru etichetarea automată a unui text. TBL reține cunoștințe lingvistice într-o formă lizibilă și trece de la o stare la altă stare folosind reguli de transformare. Aceasta combină cele 2 tipuri de taggere prezentate anterior, aceasta poate folosi tehnici de învățare automată să deducă reguli din setul de date [9].

Pentru a înțelege cum funcționează aceste sisteme se dau următorii pași:

- Se începe cu soluția – TBL de obicei începe cu soluția unei probleme și va începe să lucreze pe perioade
- Cea mai benefică transformare aleasă – în fiecare perioadă, TBL va alege cea mai benefică transformare
- Se aplică problemei – transformarea aleasă în pasul anterior va fi aplicată problemei

Algoritmul se va finaliza când transformarea aleasă în pasul 2 nu va adăuga nicio valoare nouă sau nu vor mai exista transformări. Acest tip de învățare este cel mai potrivit pentru sarcina de clasificare.

Proprietăți ale sistemelor de etichetare bazate pe transformări:

- Se învață un set mic de reguli iar aceste reguli vor fi suficiente pentru procesul de etichetare
- Depanarea este foarte ușoară deoarece regulile învățate sunt ușor de înțeles
- Complexitatea în procesul de etichetare este redus doarece TBL combină tehnicile de învațare automată și regulile generate de om
- TBL e mult mai rapid decât un model Markov pentru procesul de etichetare
- TBL nu oferă și probabilitățile tagurilor
- Dezavantajul major al acestui sistem este viteza de antrenare pe un set mare de date

2.4 Modelul Markov

În această secțiune se va introduce conceptul de model Markov cu stări ascunse (în engleză Hidden Markov Model, prescurtat HMM), acesta fiind un *model secvențial*. Un model secvențial este un model care atribuie o clasă (sau etichetă) fiecărui bloc dintr-o secvență, așadar mapând secvența de observații la o secvență de clase. HMM este de asemenea și un model probabilistic pentru un sistem de etichetare bazat pe procese stohastice, acesta calculează o distribuție a probabilităților pe secvențele posibile de clase și alege cea mai bună secvență de clase. Multe aplicații din domeniul învățării automate, prelucrării limbajului natural și bioinformatică folosesc modelul Markov cu stări ascunse.

2.4.1 Lanțuri Markov

Modelul Markov cu stări ascunse este bazat pe o formă mai avansată a lanțurilor Makov. Un lanț Markov (în engleză Markov Chain) este un model stohastic care descrie o secvență de succesiuni de evenimente posibile în care probabilitatea fiecărui eveniment depinde doar de starea la evenimentul precedent. Lanțurile Markov au fost numite după matematicianul rus *Andrey Markov.*

Un lanț Markov este un model care oferă informații despre probabilitățile unei secvențe de variabile aleatoare, fiecare valoarea poate lua valori dintr-un set oarecare. Aceste seturi pot conține cuvinte, taguri ale părților de vorbire, simboluri reprezentând de exemplu vremea, etc. [8].

Lanțul Markov vine cu următoarea ipoteză importantă „pentru a putea prezice viitorul într-o secvență de stări, tot ce contează este starea curentă”. Toate stările de dinaintea stării curente nu au niciun impact și pot fi eliminate. Formula (2.1) descrie matematic această ipoteză:

$$P(q_i = a|q_i \dots . q_{i-1}) = P(q_i = a|q_{i-1}) \quad (2.1)$$

Unde , $q_i \dots . q_{i-1}$ este secvența de stări

Un model simplu cu lanțuri Markov este reprezentat în figura 2.2:

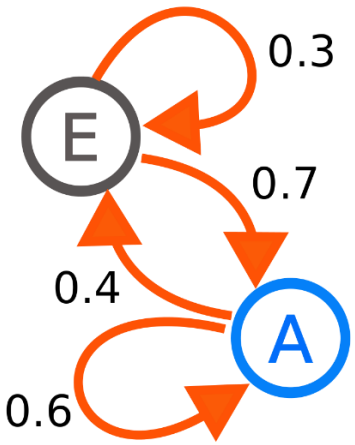

Figura 2.2 – un lanț Markov simplu [14]

În figura 2.2 există 2 stări, q_1 = E, q_2 = A și 4 lanțuri totale, acestea se numesc probabilitățile de tranziție și au valorile $a_{11} = 0.3$, $a_{12} = 0.7$, $a_{21} = 0.4$ și $a_{22} = 0.6$. De exemplu, dacă s-ar converti exemplu anterior într-un exemplu real, să zicem pentru a prezice vremea, se ia starea E ca vreme însorită și A ca vreme ploioasă. Cu presupunerea că astăzi este o zi însorită, pentru a prezice vremea de mâine, modelul indică o probabilitate de 70% ca mâine să fie o zi ploioasă.

Un lanț Markov mai poate fi vizualizat și ca un graf orientat unde stările sunt nodurile sau vârfurile iar tranzițiile sunt arcele grafului orientat.

Formal, un lanț Markov este specificat de următoarele componente:

Un set Q de N stări:

$$Q = q_1 q_2 \dots . q_N \quad (2.2)$$

Matricea A cu probabilitățile de tranziție:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \quad (2.3)$$

Distribuția inițială a probabilităților π pe stări: (aceasta este probabilitatea de a trece într-o stare pentru primul pas la timpul 0 în lanțul Markov):

$$\pi = \pi_1, \pi_2, \dots, \pi_N \quad (2.4)$$

Într-un lanț Markov, suma arcelor care pleacă dintr-o stare trebuie să fie egală cu 1:

$$\sum_{j=1}^{N} a_{ij} = 1 \; \forall \; i \tag{2.5}$$

, iar suma distribuției inițiale a probabilităților trebuie să fie egală cu 1.

$$\sum_{i=1}^{N} \pi_i = 1 \tag{2.6}$$

2.4.2 Modelul Markov cu stări ascunse (HMM)

În secțiunea anterioară, văzusem cum poate fi util un lanț Markov pentru a calcula probabilitățile unei secvențe pentru evenimente observabile. În cele mai multe cazuri, evenimentele de care suntem interesați sunt ascunse. De exemplu, părțile de vorbire într-un text nu sunt vizibile cititorului, acestea fie fiind deduse din text sau sunt cunoscute de cititor.

Un model Markov cu stări ascunse are posibilitatea de a face legătura dintre evenimentele observabile (cum ar fi cuvintele din text) și evenimentele ascunse (tagurile părților de vorbire). Înafară de componentele descrise la lanțurile Markov în formulele (2.2), (2.3), (2.4), un HMM mai are și următoarele componente:

O secvență O de T observații fiecare extras dintr-un vocabular V (acestea sunt cuvintele din text):

$$O = o_1, \; o_2, \ldots o_t, \ldots, o_T \tag{2.7}$$

O secvență B cu probabilități de observație (în engleză observation likelihoods), numită și secvența cu probabilități de emisie, în care se va exprima probabilitatea ca o observație o_t este generată dintr-o stare q_i:

$$B = b_i(o_t) \tag{2.8}$$

Un model Markov cu stări ascunse de ordinul întâi instanțiază (precum modelul cu lanțuri Markov) la rândul lui 2 ipoteze importante. Prima ipoteză este cea descrisă pentru lanțurile Markov în formula (2.1) iar cea de a doua ipoteză susține că probabilitatea unei observații de ieșire depinde doar de starea care a produs observația q_i și nu de alte stări sau alte observații. Formula (2.9) descrie matematic această ipoteză:

$$P(q_i = o_i | q_i \ldots, q_i, \ldots, q_T, o_1, \ldots o_t, \ldots, o_T) = P(q_i = o_i | q_i) \tag{2.9}$$

2.4.2.1 Componentele modelului Markov cu stări ascunse

Un model Markov cu stări ascunse are 2 componente importante, matricea A cu probabilitățile de tranziție și matricea B cu probabilitățile de emisie.

Matricea A (cu probabilitățile de tranziție) reprezintă probabilitatea ca un tag să apară după ce un alt tag a apărut la pasul anterior. De exemplu, știind că articolul „The” a apărut în pasul

precedent, cel mai probabil ca la pasul curent tagul selectat să fie substantiv „The car is ...”. Aceasta se calculează numărând secvențele de taguri.

În subcapitolul anterior *Etichetarea părților de vorbire*, am menționat la sisteme bazate pe procese stohastice, folosirea conceptului de n-gram. Acesta presupune crearea unei tabele cu frecvențele de apariție a secvențelor de n taguri. Formula matematică generală care descrie un n-gram este următoarea:

$$P(t_i|t_{i-1}, t_{i-2}, \dots, t_{i-N}) = \frac{c(t_{i-N}, \dots, t_{i-2}, t_{i-1}, t_i)}{c(t_{i-N}, \dots, t_{i-2}, t_{i-1})} \quad (2.10)$$

Unde, $P(t_i|t_{i-1}, t_{i-2}, \dots, t_{i-N}) -$ probabilitatea de tranziție a secvenței n gram $t_{i-N}, \dots, t_{i-2}, t_{i-1}, t_i$

$c(t_{i-N}, \dots, t_{i-2}, t_{i-1}, t_i) -$ frecvența de apariție a secvenței n gram $t_{i-N}, \dots, t_{i-2}, t_{i-1}, t_i$

$c(t_{i-N}, \dots, t_{i-2}, t_{i-1}) -$ frecvența de apariție a secvenței n gram fără tagul curent $t_{i-N}, \dots, t_{i-2}, t_{i-1}$

Pentru a se putea forma o tabelă cu frecvențele de apariție pentru un n-gram, modelul Markov trebuie să folosească un text din care să extragă aceste probabilități de tranziție, acest text se va numi set de antrenament. Pentru a colecta frecvențele de apariție, fiecare secvență din setul de antrenament va fi adăugată sau incrementată în matricea A.

Deobicei, într-un sistem de etichetare cu un model Markov cu stări ascunse se calculează doar probabilitățile de tranziție pentru unigram (1-gram), bigram (2-gram) și trigram (3-gram). Unigramul nu se uită la niciun tag anterior, deobicei acesta nu este utilizat direct într-un sistem de etichetare ci este folosit când celelalte n-gramuri nu dețin destule informații:

$$P(t_i) = \frac{c(t_i)}{N} \quad (2.11)$$

Unde, $P(t_i) -$ probabilitatea unigramului pentru tagul t_i

$c(t_i) -$ frecvența de apariție a tagului t_i

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

, bigramul se uită doar la tagul precedent:

$$P(t_i|t_{i-1}) = \frac{c(t_{i-1}, t_i)}{c(t_{i-1})} \quad (2.12)$$

Unde, $P(t_i|t_{i-1}) -$ probabilitatea bigramului pentru secvența de taguri t_{i-1}, t_i

$c(t_{i-1}, t_i) -$ frecvența de apariție pentru secvența bigram ale tagurilor t_{i-1}, t_i

$c(t_{i-1}) -$ frecvența de apariție a tagului t_{i-1}

, iar trigramul, fiind n-gramul cel mai avansat dintre cele 3 menționate, se uită la ultimele 2 taguri precedente:

$$P(t_i|t_{i-1}, t_{i-2}) = \frac{c(t_{i-2}, t_{i-1}, t_i)}{c(t_{i-2}, t_{i-1})} \quad (2.13)$$

Unde, $P(t_i|t_{i-1}, t_{i-2})$– probabilitatea trigramului pentru secvența de taguri t_{i-2}, t_{i-1}, t_i

$c(t_{i-2}, t_{i-1}, t_i)$ – frecvența de apariție pentru secvența trigram ale tagurilor t_{i-2}, t_{i-1}, t_i

$c(t_{i-2}, t_{i-1})$ – frecvența de apariție pentru secvența bigram ale tagurilor t_{i-2}, t_{i-1}

Cu cât crește rangul n-gramului selectat, cu atât unele secvențe de probabilități pot lipsi, secvențele lipsă apar cel mai des la modelul trigram. Pentru a rezolva această problemă se utilizează diverse tehnici de netezire a datelor, astfel încât atunci când nu este găsită o secvență n-gram, valoarea ei este dată de o valoare implicită.

O metodă de netezire a datelor este interpolarea liniară. Aceasta presupune calcularea unei noi probabilități compuse din suma probabilităților de tranziție (unigram, bigram, trigram) înmulțite cu o pondere determinată:

$$P_{LI}(t_3|t_1, t_2) = \lambda_1 P(t_3) + \lambda_2 P(t_3|t_2) + \lambda_3 P(t_3|t_1, t_2) \tag{2.14}$$

Valorile ponderilor $\lambda_1, \lambda_2, \lambda_3$ sunt estimate prin *interpolarea eliminată*. Pseudocodul general de determinare a acestor ponderi este următorul:

```
set λ1 = λ2 = λ3 = 0
foreach trigram t1, t2, t3 with f(t1, t2, t3) > 0
    depending on the maximum of the following three values:
        case (f(t1,t2,t3)−1)/(f(t1,t2)−1): increment λ3 by f(t1, t2, t3)
        case (f(t2,t3)−1)/(f(t2)−1): increment λ2 by f(t1, t2, t3)
        case (f(t3)−1)/(N−1): increment λ1 by f(t1, t2, t3)
    end
end
normalize λ1, λ2, λ3
```

Figura 2.3 – pseudocodul pentru interpolarea liniară (Deleted Interpolation) a trigramului [8]

O altă metodă de netezire este netezirea aditivă [18]. Aceasta presupune adunarea la numărător cu o constantă aleasă de dinainte și la numitor adunarea cu produsul dintre această constantă și o altă valoare care reprezintă lungimea setului de date x.

$$\theta_i = \frac{x_i + \alpha}{N + \alpha d} \tag{2.15}$$

Unde, $i = 1, \ldots, d$

Pentru $\alpha = 0$, atunci formula (2.15) nu folosește nicio netezire iar pentru $\alpha = 1$ atunci noua formulă se va numi *regula de succesiune a lui Laplace* sau *formula de netezire a lui Laplace*.

Matricea B cu probabilitățile de emisie reprezintă probabilitatea ca un anumit tag să fie asociat cu un anumit cuvânt din setul de antrenament. Formula (2.8) descrie general această probabilitate de observații dar formula folosită de sistem care descrie estimarea maximă a probabilității este următoarea:

$$P(w_i|t_i) = \frac{c(t_i, w_i)}{c(t_i)} \tag{2.16}$$

Unde, $P(w_i|t_i)$ – probabilitatea de emisie a cuvântului w_i asociat cu tagul t_i
$c(t_i, w_i)$ – frecvența de apariție a cuvântului w_i asociat cu tagul t_i
$c(t_i)$ – frecvența de apariție a tagului t_i

HMM, aici, este un algoritm de învățare supervizată pe mai multe clase deoarece modelul folosește un set de antrenament care este deja etichetat cu tagurile corecte. Algoritmul clasifică fiecare token (cuvânt) din set cu clasa asociată lui, clasele aici fiind tagurile părților de vorbire, setul de taguri având o dimensiune mai mare de 2 taguri totale (multiclass classification).

Pentru exemplul următor:

„Cristopher(Noun) Nolan(Noun) can(Modal Verb) hire(Verb) Will(Noun). Tip(Noun) will(Modal Verb) hire(Verb) Cristopher(Noun). Will(Modal Verb) Nolan(Noun) tip(Verb) Cristopher(Noun)? Cristopher(Noun) will(Modal Verb) pay(VB) Tip(Noun).”

, modelul Markov cu stări ascunse este acesta:

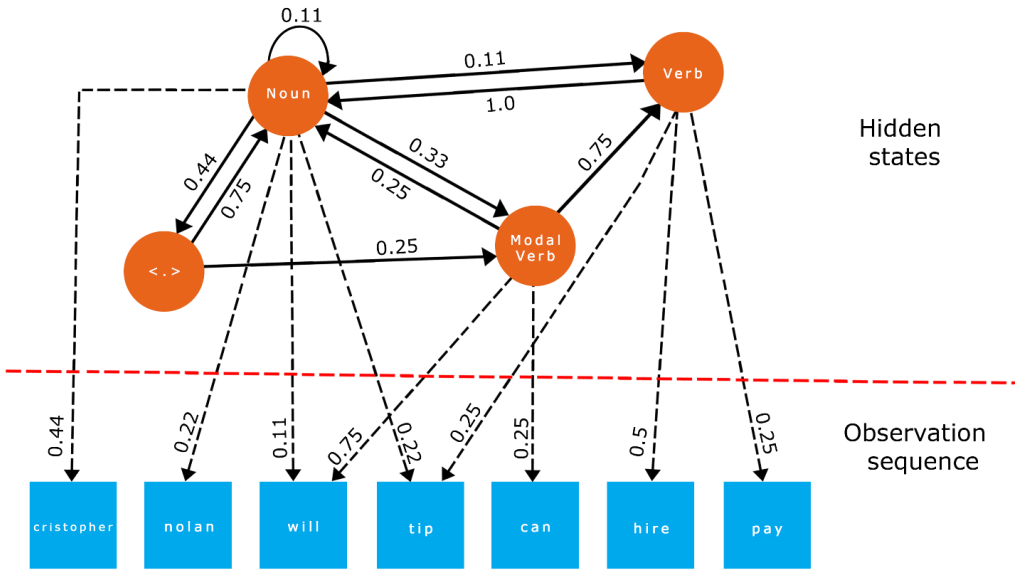

Figura 2.4 – exemplu model Markov cu stări ascunse

2.4.2.2 Decodificarea modelului Markov cu stări ascunse

Pentru orice model, precum HMM, care conține variabile ascunse, sarcina de a determina secvența variabilelor ascunse Q corespunzătoare secvenței de observații O, se numește **decodificare**.

Pentru etichetarea părților de vorbire, scopul operației de decodificare este de a alege cea mai probabilă secvență de taguri, dându-se secvența de observații.

$$\hat{t}_1^n = argmax^{t_1^n} P(w_1^n|t_1^n) \tag{2.17}$$

HMM presupune 2 ipoteze simple. Prima ipoteză susține că probabilitatea ca un cuvânt să apară depinde doar de tagul acestuia și e independent de tagurile & cuvintele vecine (aceasta calculează probabilitatea de emisie):

$$P(w_1^n|t_1^n) \cong \prod_{i=1}^{n} P(w_i|t_i) \tag{2.18}$$

A doua ipoteză, numită și ipoteza bigram, susține că probabilitatea unui tag este dependentă doar de tagul precendent, ci nu de întreaga secvență (aceasta calculează probabilitatea de tranziție):

$$P(t_1^n) \cong \prod_{i=1}^{n} P(t_i|t_{i-1}) \tag{2.19}$$

Combinând formulele matematice (2.17), (2.18), (2.19), rezultă următoarea formulă care obține secvența de taguri cu cea mai mai mare probabilitate pentru un model de tip bigram:

$$\hat{I}_1^n = argmax_{t_1^n} P(w_1^n|t_1^n) \cong argmax_{t_1^n} \prod_{i=1}^{n} \overbrace{P(w_i|t_i)}^{emission} \overbrace{P(t_i|t_{i-1})}^{transition} \tag{2.20}$$

Cele 2 părți din ecuația (2.20) corespund matricii A cu probabilitățile de tranziție și matricii B cu probabilitățile de emisie, definite anterior.

2.4.2.3 Algoritmul Viterbi

Algoritmul Viterbi, numit și Viterbi path în engleză, este un algoritm de programare dinamică pentru a găsi cea mai probabilă secvență în stările ascunse. Acesta este extrem de folosit pentru recunoașterea vorbirii și în domenii ca prelucrarea limbajului natural și bioinformatică.

Algoritmul a fost numit după inginerul american *Andrew Viterbi*, care l-a propus în 1967, ca algoritm de decodificare în codurile de convoluție pentru legăturile din comunicația digitală cu zgomot. Algoritmul Viterbi a devenit un termen standard pentru aplicațile cu algoritmi de programare dinamică care urmăresc maximizarea problemelor folosind probabilități [15].

Pseudocodul pentru algoritmul Viterbi este următorul:

```
function VITERBI(O, S, Π, Y, A, B) : X
    for each state i = 1, 2, ..., K do
        T1[i, 1] ← π_i · B_{i y_1}
        T2[i, 1] ← 0
    end for
    for each observation j = 2, 3, ..., T do
        for each state i = 1, 2, ..., K do
            T1[i, j] ← max_k (T1[k, j − 1] · A_ki · B_{i y_j})
            T2[i, j] ← arg max_k (T1[k, j − 1] · A_ki · B_{i y_j})
        end for
    end for
    z_T ← arg max_k (T1[k, T])
    x_T ← s_{z_T}
    for j = T, T − 1, ..., 2 do
        z_{j−1} ← T2[z_j, j]
        x_{j−1} ← s_{z_{j−1}}
    end for
    return X
end function
```

Figura 2.5 - Pseudocod Viterbi - wiki [15]

Complexitatea algoritmului este $O(T\ x\ |S|^2)$.

Pseudocodul adaptat la exemplul pentru etichetarea părții de vorbire este următorul:

function VITERBI(*observations* of len *T*,*state-graph* of len *N*) **returns** *best-path*, *path-prob*

create a path probability matrix *viterbi[N,T]*
for each state *s* **from** 1 **to** *N* **do** ; initialization step
  $viterbi[s,1] \leftarrow \pi_s * b_s(o_1)$
  $backpointer[s,1] \leftarrow 0$
for each time step *t* **from** 2 **to** *T* **do** ; recursion step
  for each state *s* **from** 1 **to** *N* **do**
    $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
    $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
$bestpathprob \leftarrow \max_{s=1}^{N} viterbi[s,T]$; termination step
$bestpathpointer \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T]$; termination step
bestpath ← the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
return *bestpath*, *bestpathprob*

Figura 2.6 – Pseudocod Viterbi pentru etichetarea părţii de vorbire [8]

Acesta mai întâi va crea o matrice de probabilități, cu o coloană pentru fiecare observație o_t, și o linie pentru fiecare stare din graficul de stări. Fiecare coloană va avea o celulă pentru fiecare stare q_i, ca în trellis-ul [20] din figura 2.7. În primul pas, $v_t(j)$ se va calcula doar tranziția de la starea inițială la starea curentă (de exemplu, tranziția de la începutul propoziției la un tag de substantiv) înmulțit cu probabilitatea de emisie pentru prima stare de observație (primul cuvânt). Fiecare celulă din matrice, $v_t(j)$, reprezintă probabilitatea ca modelul Markov cu stări ascunse să fie în starea j după ce a vazut primele t stări de observație. Valoarea pentru fiecare celulă $v_t(j)$ este calculată recursiv luând calea cea mai probabilă care ar putea duce spre această celulă. Formal, fiecare celulă exprimă probabilitatea următoare [8]:

$$v_t(j) = max^{q_1 \dots q_{t-1}} P(q_1 \dots . q_{t-1}, o_1 \dots . o_t, q_t = j|\lambda) \quad (2.21)$$

Calea cu probabilitatea cea mai mare este luată ca maximul dintre toate stările precedente. După ce s-a calculat probabilitatea pentru fiecare stare la timpul t = i – 1 (starea precedentă), probabilitatea Viterbi pentru celula curentă se va actualiza pe baza valorii celei mai mari care duce spre celula curentă. Formula matematică pentru calcularea acestei probabilități este:

$$v_t(j) = max_{i=1}^{N} v_{t-1}(i) a_{ij} b_j(o_t) \quad (2.22)$$

unde, $v_t(j) -$ probabilitatea celulei curente

$v_{t-1}(i) -$ probabilitatea celulei procesate la pasul de timp anterior

a_{ij} – probabilitatea de tranziție de la starea (tagul) anterioară q_i la starea curentă q_j

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

Când algoritmul ajunge la celula finală, se va face backtracking până la celula inițială și se vor emite clasele sau tagurile cu probabilitatea cea mai mare pentru fiecare observație (cuvânt) din setul folosit pentru decodificare (deobicei numit setul de testare).

Exemplu de decodificare cu algoritmul Viterbi pentru secvența „Nolan will tip Will”:

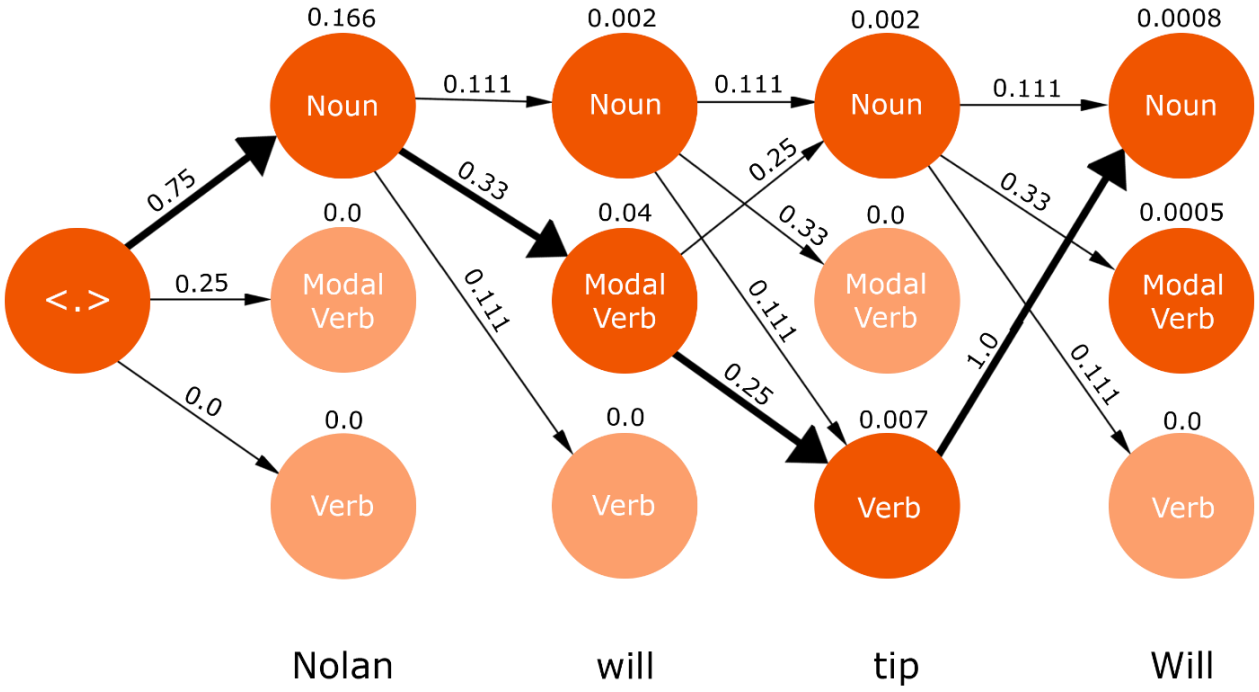

Figura 2.7 – stările Trellis-ului pentru modelul din figura 2.4

2.4.3 Cuvinte necunoscute

Pentru a ajunge la rezultate foarte bune, în sistemele de etichetare a părților de vorbire, este important să existe un model care să se ocupe cu tratarea cuvintelor necunoscute. Cuvinte noi precum nume, substantive comune, verbe, acronime, apar destul de des în limba engleză iar un set de antrenament nu ar putea să le cuprindă pe toate.

Un indicator folositor care ar putea distinge dintre părțile de vorbire a unui cuvânt necunoscut este acela de a verifica forma cuvântului, dacă un cuvânt începe cu literă mare atunci cel mai probabil acel cuvânt are tagul de substantiv propriu. Cel mai puternic indicator pentru a indica partea de vorbire a unui cuvânt necunoscut este morfologia. Acest indicator verifică sufixele (aici se referă la secvențele de caracter cu care se termină un cuvânt) pentru a deduce partea de vorbire. De exemplu, cuvintele care se termină în -s au o probabilitate mai mare să fie substantive, cuvintele care se termină în -ed tind să fie verbe, cele care se termină în -able în majoritatea timpului sunt adjectiv, etc. În limba engleză, sufixele și prefixele sunt un indicator foarte bun pentru a deduce partea de vorbire a cuvintelor necunoscute, de exemplu în ziarul american „Wall Street Journal”, cuvintele care se termină cu “able” sunt etichetate ca adjective în 98% din cazuri [19].

Modelul prezentat în această lucrare își propune să adune o listă de sufixe și de prefixe, după care să calculeze probabilitatea fiecărui tag asociat cu acestea. Sufixele & prefixele sunt, de

asemenea, separat calculate pentru cuvintele care încep cu literă mare și cuvintele care încep cu literă mică. Formula generală de calcul a probabilității pentru sufixe și prefixe este următoarea:

$$P_{sp}(x_i|t_i) = \frac{c(t_i, x_i)}{\sum_{k=1}^{T_n^{x_i}} k} \tag{2.23}$$

Unde, $P_{sp}(x_i|t_i)$ – probabilitatea de asociere a unui sufix/prefix x_i, cu tagul t_i

$c(t_i, x_i)$ – frecvența de apariție a sufixului/prefixului x_i cu tagul t_i

$\sum_{k=1}^{T_n^{x_i}} k$ – suma tuturor tagurilor asociate sufixului/prefixului x_i

La implementare, această formulă va fi netezită pentru sufixele și prefixele care nu apar în setul de antrenament. Detalii despre procesul de netezire pentru formula (2.23) se vor descrie mai detaliat în capitolul următor.

Altă metodă care se poate folosi pentru a deduce partea de vorbire a cuvintelor necunoscute este utilizarea unor ponderi care se vor calcula pe baza unor reguli scrise manual sau deduse din setul de antrenament. Aceste reguli pot verifica dacă un cuvânt începe cu literă mare, ce caractere speciale (precum punct, linie, bară) apar & cum afectează acestea un cuvânt, dacă acesta se termină sau începe cu diferite caractere care indică un tag special, etc.

De asemenea, modelul poate folosi 2 tipuri de matricii pentru a colecta probabilitățile de emisie. Una poate colecta probabilitățile pentru cuvintele care încep cu literă mare iar cealaltă va colecta probabilitatea de emisie a cuvintelor care fie încep cu literă mică, fie a tuturor cuvintelor convertite la literă mică. Acest lucru s-a dovedit a fi foarte util pentru procesul de dezambiguizare pentru diferite seturi de date din limba engleză [19].

Modelul implementat în această lucrare folosește probabilitățile sufixelor/prefixelor dar și un set de reguli scris manual. Modelul de tratare a cuvintelor necunoscute calculează și returnează probabilitatea de asociere a unui cuvânt de intrare cu fiecare tag existent în setul de antrenament. Modelul va calcula probabilitatea pentru fiecare tag iar tagul care are cea mai mare probabilitate va fi ales ca tag pentru cuvântul necunoscut. Procesul și formula de calcul a probabilității respective vor fi descrise detaliat în capitolul următor.

2.5 Evaluarea algoritmilor de învățare

O metrică de evaluare foarte importantă pentru învățarea supervizată sau clasificarea cuvintelor cu partea lor de vorbire, este calculul acurateței. Aceasta se poate calcula fie prin metoda simplă care presupune următoarea formulă:

$$Acc = \frac{numărul\ de\ cazuri\ corect\ identificate}{numărul\ total\ de\ cazuri} \tag{2.24}$$

, fie prin formula (2.26) care folosește matricea de eroare (în engleză confusion matrix).

Pentru acest tip de problemă (etichetarea părților de vorbire), se poate calcula acuratețea totală dar și acuratețea pentru cuvintele cunoscute și pentru cuvintele necunoscute. Cuvintele cunoscute presupun, aici, cuvintele care apar în setul de antrenament iar cuvintele necunoscute sunt cuvintele care nu apar în setul de antrenament. Acuratețea pe cuvintele necunoscute verifică performanțele sistemului de etichetare atunci când întâlnește cazuri noi și trebuie să se adapteze situațiilor necunoscute.

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) [16]. 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 rezultatul total pe o metrică este calculat ca fiind media aritmetică a rezultatelor pe toate clasele (tagurile).

$$\mu_{Total} = \frac{1}{N} \cdot \sum_{i=1}^{N} x_i \tag{2.25}$$

N – numărul de clase (taguri) individuale în setul de antrenament

x_i – rezultatul statistic pentru clasa/tagul x_i

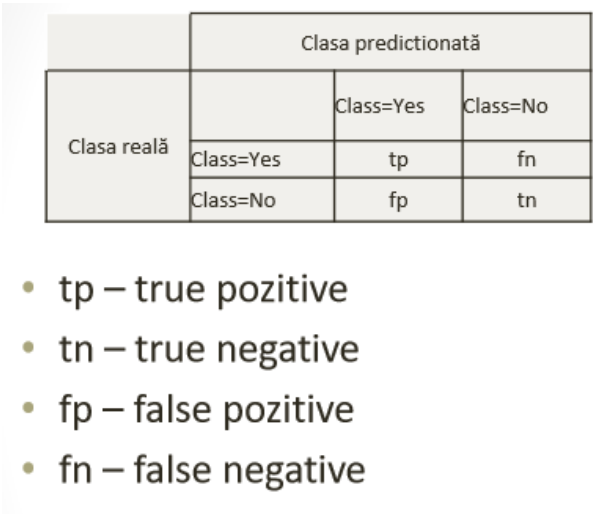

Figura 2.8 – matricea de eroare – informații [17]

Fiecare clasă unică din setul de testare va avea propria ei matrice de eroare, 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 calcula în urma utilizării unei matricii de eroare sunt:

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

$$A = \frac{tp + tn}{tp + tn + fp + fn} \tag{2.26}$$

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

$$P = \frac{tp}{tp + fp} \tag{2.27}$$

Recall (True positive rate): 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?”

$$R = \frac{tp}{tp + fn} \tag{2.28}$$

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

$$S = \frac{tn}{tn + fp} \tag{2.29}$$

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

$$F = (1 + \beta^2)\frac{P * R}{\beta^2 * P + R} \tag{2.30}$$

Deobicei, parametrul β are valoarea 1, F-measure devenind astfel **F1-Score**, formula (2.30) fiind redusă la forma următoare:

$$F_1 = 2\frac{P * R}{P + R} \tag{2.31}$$

III Consideraţii practice

3.1 Descriere generală proiect

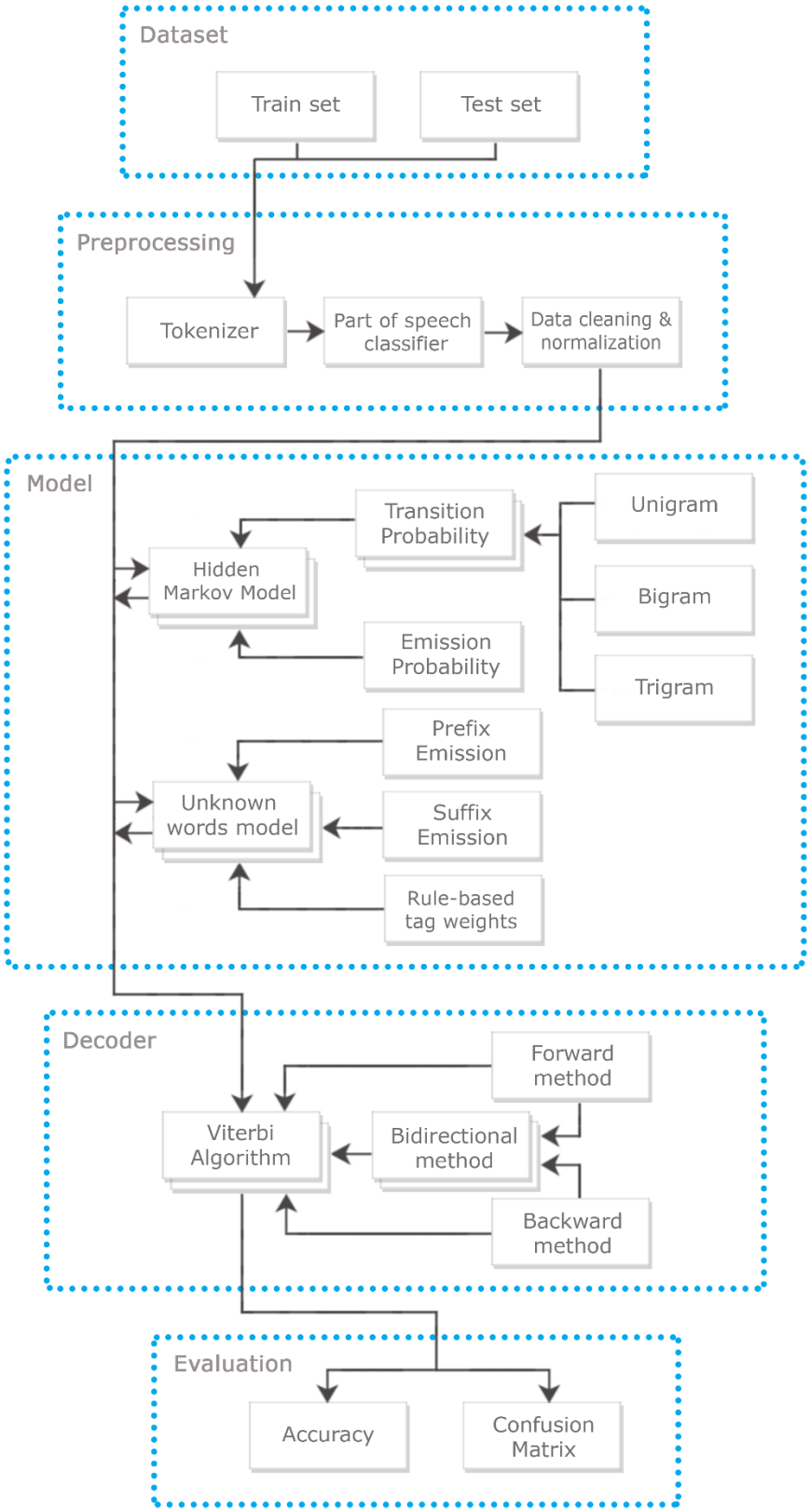

Figura 3.1 - Arhitectura sistemului de etichetare

În acest capitol se va prezenta arhitectura sistemului de etichetare a părților de vorbire propus în această lucrare. Sistemul de etichetare prezentat în acest capitol a fost antrenat și conceput să lucreze doar cu texte din limba engleză. Sistemul poate fi antrenat și pe texte în altă limbă dar rezultatele nu vor fi la fel de bune.

Se poate observa în figura 3.1 că sistemul este structurat pe 5 mari componente: *Dataset, Preprocessing, Model, Decoder* și *Evaluation*. În prima componentă, Dataset, este descris setul de date pe care se antrenează algoritmul de învățare și pe care este testat acesta, pentru a obține performanțele sistemului de etichetare. În acesta, se descriu și 2 metode de antrenare diferite, una bazată pe antrenare și testare cu o împărțire fixă între acestea, 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 conțină informații eronate sau false. Această componentă mai include 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 modelul Markov cu stări ascunse (HMM) și modelul pentru cuvintele necunoscute. Modelul este cea mai importantă componentă din sistem, fără aceasta nu s-ar putea deduce partea de vorbire a cuvintelor. În Decoder, este descris algoritmul recursiv Viterbi de programare dinamică și metodele implementate împreună cu acest algoritm.

În ultima componentă, Evaluation, este evaluat algoritmul de predicție (clasificare), în funcție de diferitele metrici de evaluare precum acuratețea, precizia, recall-ul, f-measure-ul și specificitatea.

Proiectul a fost scris în limbajul de programare C# .NET, target framework: .NET Core 2.1. Pe pagina de github a proiectului [21], 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. În următorul subcapitol, se vor descrie detaliat fiecare componentă și subcomponentă a ei & codul C# specific fiecărei secțiuni.

3.2 Dezvoltare aplicație

3.2.1 Blocul 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 unui sistem de etichetare a părților de vorbire. 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 [22]. 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, cărți, muzică, dans) – 17 documente

D. Religie – 17 documente

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

F. Folclor popular – 48 documente

G. Scrisori, bibliografii, biografii – 75 documente

H. Diverse – 30 documente

J. Articole științifice– 80 documente

TOTAL – 374 documente

, iar a doua categorie fiind proză imaginativă cu urmatoarele subcategorii:

K. Ficțiune generală – 29 documente

L. Mister și ficțiune detectivă – 24 documente

M. Opere științifico-fantastice – 6 documente

N. Aventură și ficțiune western – 29 documente

P. Povești de dragoste – 29 documente

R. Umor – 9 documente

TOTAL – 126 documente

Fiecare document are peste 2000 de cuvinte iar fiecare cuvânt (numit și token) este delimitat de un slash '/', urmat de tagul aferent părții de vorbire al acestuia, sub forma "token/tag". Setul de taguri folosit este **Penn Treebank**, acesta conține 45 de taguri:

Tag	Description	Example	Tag	Description	Example	Tag	Description	Example
CC	coordinating conjunction	*and, but, or*	PDT	predeterminer	*all, both*	VBP	verb non-3sg present	*eat*
CD	cardinal number	*one, two*	POS	possessive ending	*'s*	VBZ	verb 3sg pres	*eats*
DT	determiner	*a, the*	PRP	personal pronoun	*I, you, he*	WDT	wh-determ.	*which, that*
EX	existential 'there'	*there*	PRP$	possess. pronoun	*your, one's*	WP	wh-pronoun	*what, who*
FW	foreign word	*mea culpa*	RB	adverb	*quickly*	WP$	wh-possess.	*whose*
IN	preposition/ subordin-conj	*of, in, by*	RBR	comparative adverb	*faster*	WRB	wh-adverb	*how, where*
JJ	adjective	*yellow*	RBS	superlatv. adverb	*fastest*	$	dollar sign	$
JJR	comparative adj	*bigger*	RP	particle	*up, off*	#	pound sign	#
JJS	superlative adj	*wildest*	SYM	symbol	+,%, &	"	left quote	' or "
LS	list item marker	*1, 2, One*	TO	"to"	*to*	"	right quote	' or "
MD	modal	*can, should*	UH	interjection	*ah, oops*	(	left paren	[, (, {, <
NN	sing or mass noun	*llama*	VB	verb base form	*eat*	)	right paren	],), }, >
NNS	noun, plural	*llamas*	VBD	verb past tense	*ate*	,	comma	,
NNP	proper noun, sing.	*IBM*	VBG	verb gerund	*eating*	.	sent-end punc	. ! ?
NNPS	proper noun, plu.	*Carolinas*	VBN	verb past part.	*eaten*	:	sent-mid punc	: ; ... – -

Figura 3.2 – Penn Treebank tagset [8]

Acest set de date este folositor pentru un algoritm de *învățare supervizată pe mai multe clase* sau *multiclass classification* în engleză. Exemplu de propoziție din 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 ./.*

În Brown Corpus, există peste 100 de taguri (~103) individuale, multe fiind derivate de la forma de bază 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 ce pot apărea în setul de date, pot fi formate din tagul propriu-zis și un tag prefix de indicație a unei informații suplimentare, folosind delimitatorul '-'.

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

, unde */fw-in, /fw-nn* sunt tagurile de prepoziție, respectiv substantiv la singular, pentru cuvintele „en route" din limba franceză (fw – foreign word).

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

3.2.1.1 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 setul de date trebuie împărțit într-un set de antrenament (train set) și un set de testare (test set). Există 2 metode de împărțire a setului de date folosite în proiect:

3.2.1.1.1 Împărțire 70% - setul de antrenament, 30% - setul de testare

Această metodă nu necesită un algoritm de împărțire, 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. subcategoria *J. Articole științifice,* are în total 80 de documente, primele 56 documente (70%) vor fi folosite în etapa de antrenare iar ultimele 24 documente (30%) vor fi folosite în etapa de testare.

3.2.1.1.2 Cross-Validation

Numită și "rotation estimation", este o tehnică de validare pentru a se vizualiza rezultatul generalizat al modelului pentru un set de date independent. Acest mod de validare este folositor pentru a vizualiza 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 date necunoscute [23].

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 seturi, acestea fiind numite folds. După această împărțire, se vor itera foldurile create, astfel încât fiecare fold va deveni la rândul lui set de testare iar restul foldurilor vor deveni set de antrenare. Se va salva acuratețea modelului pentru fiecare fold în parte, în final obținând un rezultat pentru toate aceste folduri. În figura 3.3 se poate observa procesul de crossvalidation pentru K = 4.

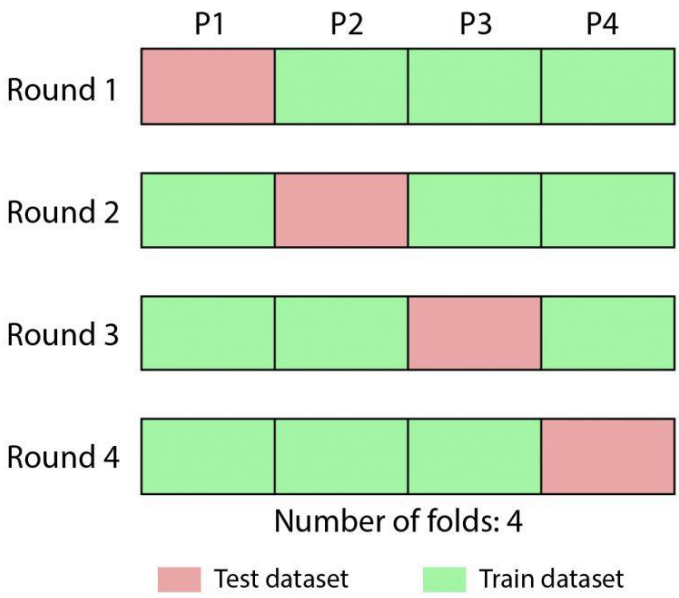

Figura 3.3 – exemplu cross-validation [24]

La final, după ce s-a evaluat fiecare fold în parte, se va calcula media aritmetică pe folduri pentru a obține o acuratețe totală pe tot setul de date:

$$\mu = \frac{1}{k} \cdot \sum_{i=1}^{k} x_i \tag{3.1}$$

k - numărul de folduri alese

x_i – 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 ajunge la acesta, ignorând elementele din listă deja folosite la amestecare (scratched elements). 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 [25].

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 folosite. 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 Blocul de preprocesare

Preprocesarea este procesul în care datele sunt curățate și normalizate, aceasta fiind cea mai importantă etapă într-un proiect care lucrează cu algoritmi de învățare automată (machine learning). Lipsa acestui proces poate îngreuna procesul de învățare și poate influența negativ calitatea rezultatelor, deseori algoritmul de învățare returnând rezultate false fără o etapă de preprocesare. Cele 3 subprocese implementate în etapa de preprocesare aici, sunt:

- Tokenizarea
- Clasificator părți de vorbire
- Curățarea și normalizarea datelor

3.2.2.1 Tokenizarea

Procesul de a delimita cuvintele dintr-un text și posibil a le clasifica, folosit în analiza lexicală, se numește proces de tokenizare. Acesta deobicei delimitează pe baza unei reguli de despărțire, algoritmul de tokenizare folosit de sistemul de etichetare se numește *Whitespace Tokenizer*.

Algoritmul *Whitespace Tokenizer* împarte un string într-o listă de stringuri, 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 Whitespace Tokenizer

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 token-uri va fi de forma: list = […., “he/pn”, “made/vbd”, “pancakes/nn”, “./.” , ….].

Pentru a putea lucra cu tokenul și tagul separat, se preferă despărțirea cuvântului de tag, pentru asta se implementează următoarea structură:

```
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 '/' iar cuvântul și tagul acestuia vor fi adăugate într-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 favorabil, există un vector de stringuri de dimensiune = 2 (cuvânt necompus+tag) iar în cazul opus dimensiunea vectorului va fi mai mare de 2. Pentru asta, se iterează vectorul de cuvinte și se concatenează tot ce apare în vector până la tag. La final, se va adăuga în lista de structuri, tokenul rezultat în urma procesului de concatenare și tagul acestui token.

3.2.2.2 Clasificator părți de vorbire

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, predicția ar avea de suferit. Pentru a rezolva această problemă, se introduce un clasificator al părților de vorbire care va clasifica fiecare parte de vorbire din setul de date în 10 categorii, acestea fiind părțile de vorbire de bază din limba engleză. Acest proces nu este unul automat realizat de un algoritm, ci este realizat în urma unei analize a părților de vorbire utilizate în limba engleză [26], [27], [28], [29]. S-au ales următoarele categorii:

1) Noun (NN) – substantiv
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) – articol & determinant (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, numere cardinale ("six", "two"), cuvinte din altă limbă, etc.

Orice cuvânt etichetat din setul de date va avea, după acest proces, doar un singur tag din cele 10 menționate anterior. În urma clasificării, s-au realizat și diferite statistici pentru distribuția tagurilor din setul de date. Acestea se pot vizualiza în figurile 3.4 și 3.5:

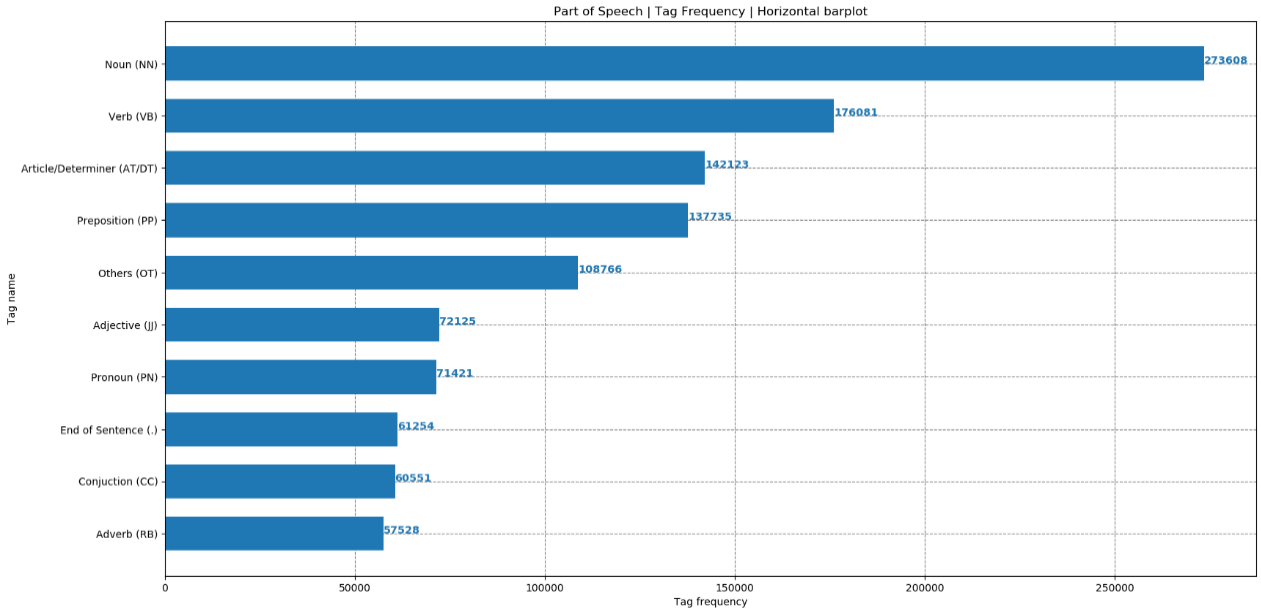

Figura 3.4 – Frecvența noilor taguri din setul de date

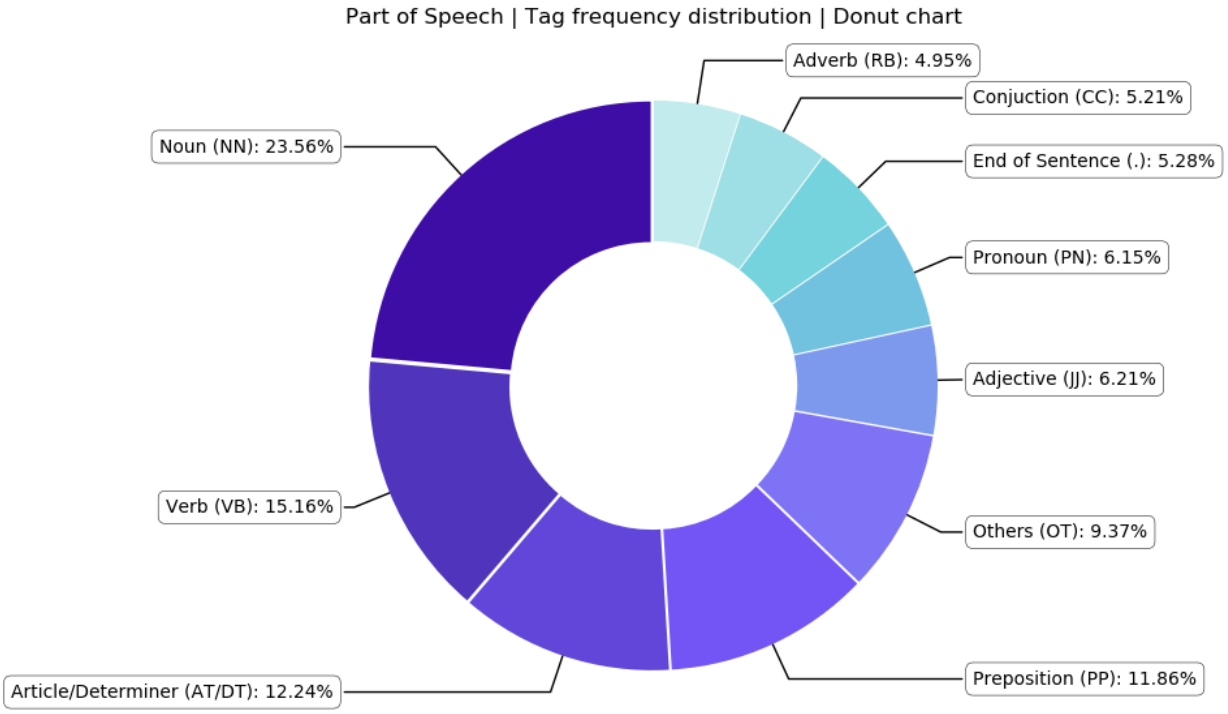

Figura 3.5 – Procentul pentru fiecare tag din setul de date (%)

Se observă că aproape un sfert de cuvinte din setul de date sunt substantive, dacă s-ar crea un sistem de etichetare simplu care returnează doar tagul de substantiv, acesta ar putea prezice corect 23.56% de cuvinte din setul de date.

Implementarea clasificatorului

Procedeul principal este de a itera lista de taguri și de a crea o nouă listă în care vechile taguri vor face parte dintr-una din cele 10 categorii descrise anterior.

Acesta returnează un index în funcție de unde se află tagul 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 tagul iterat există în lista BrownCorpusTags, dacă există se va returna index-ul din listă, altfel 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.

```
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;
```

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 este un caz special deoarece, funcția găsește secvența cu pattern-ul “ql” înainte de a găsi “wql” în listă (“ql” este catalogat ca adverb iar dacă va căuta cu funcția Contains atunci “wql” va fi catalogat ca adverb când ar trebuii să fie conjuncție). 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 Curățarea și normalizarea datelor

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 afecteze negativ procesul de învățare.

Această componentă va elimina mai întâi cuvintele de oprire (caracterele care nu sunt importante la etichetarea părților 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 elimină cifrele din acesta iar dacă trece de un anume prag de litere rămase atunci tokenul nu va fi eliminat. Înainte ca preprocesarea să se finalizeze, în etapa de antrenare, se va păstra și o listă de cuvinte care încep cu literă mare și încă o listă în care toate cuvintele sunt converite la literă mică. Cuvintele din setul de testare vor trece prin același filtru, cu excepția ultimei etape, cuvintele care încep cu literă mare nu vor fi converite la literă mică.

În setul de testare, de asemenea, se vor elimina repetițiile pentru token-urile care au tag de sfârșit/început de propoziție, deoarece acestea nu sunt evaluate de predictor și repetițiile de genul *„ ?/. ./. ?/. !/.”* pot emite erori dacă nu sunt tratate.

Implementare procesului de curățare și normalizare a datelor

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** (folosită în codul subliniat) din C#, în locurile unde apare o cifră (d = digit) într-un token, acestea vor fi înlocuite cu un caracter gol sau null. Funcția ToLowerCaseNormalization(string) este echivalentă cu funcția ToLower() din C#.

3.2.3 Blocul Model

Modelul este partea principală din sistem, acesta conține diferite informații despre ponderile și probabilitățile tagurilor, calculate pe baza datelor antrenate. El este format din 2 submodele:

- Modelul Markov cu stări ascunse (Hidden Markov model)
- Modelul pentru cuvintele necunoscute (Unknown words model)

3.2.3.1 Modelul Markov cu stări ascunse

Modelul Markov cu stări ascunse (HMM) este un model stohastic (folosește metode probabilistice pentru a predicționa tagurile) și este implementat după modelul clasic prezentat în capitolul *Considerații teoretice*. Acesta folosește probabilitățile de emisie și de tranziție interpolate pentru a deduce tagul cuvintelor dintr-o propoziție.

3.2.3.1.1 Probabilitatea de emisie

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

$$P(w_i|t_i) = \frac{c(t_i, w_i)}{c(t_i)} \tag{3.2}$$

Unde, $P(w_i|t_i)$ – probabilitatea de emisie a cuvântului w_i asociat cu tagul t_i
$c(t_i, w_i)$ – frecvența de apariție a cuvântului w_i asociat cu tagul t_i
$c(t_i)$ – frecvența de apariție a tagului t_i

ex. … ***he/???*** went/VB home/NN …

$$P(he|PN) = \frac{c(PN, he)}{c(PN)} = \frac{2848}{2890} \cong 0.98 \tag{3.3}$$

Putem vedea în exemplul (3.3) cum formula pentru probabilitatea de emisie este o generalizare de la formula 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’?’ [8]. Valorile din acest exemplu au fost alese aleator pentru demonstrație.

Implementarea probabilității de emisie

Pentru a putea fi aplicată formula (3.2), mai întâi trebuie să se obțină frecvența de apariție a tuturor tagurilor din setul de antrenare iar după aceea, trebuie obținută 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 crea 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 *Curățarea și normalizarea datelor* că se va păstra o listă separată pentru cuvintele care încep cu literă mare și încă o listă cu toate cuvintele converite la literă mică. Codul care colectează frecvența de apariție a tagurilor în lista de cuvinte care încep cu literă mare, 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** [30] pentru a căuta dacă în lista cu frecvențe există deja cuvântul care este iterat (Find(x => x.Word == w.word)). În cazul în care cuvântul nu este găsit în listă atunci se creează un obiect de tip EmissionModel și se adaugă cuvântul și tagul aferent acestuia în lista WordCapitalizedTagsEmissionFrequence cu un count = 1. Metoda LINQ FirstOrDefault va returna primul obiect din listă/dicționar care îndeplinește condiția expresiei lambda sau va returna null/0 în cazul în care niciun obiect din listă/dicționar nu îndeplinește condiția impusă. 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 proces se aplică si pentru cuvintele convertite la literă mică.

Odată ce s-a colectat frecvența de apariție a tuturor tagurilor fiecărui cuvânt, se poate aplica formula (3.2) doar pentru cuvintele care sunt în setul de testare dar sunt și în setul de antrenament. De remarcat că nu se calculează probabilitatea tuturor cuvintelor din setul de antrenament, ci doar acelora care sunt întâlnite în setul de testare. 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 convertite 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 & se află în lista de antrenare, atunci se continuă procesul de iterație pentru toate tagurile al acestui cuvânt, și se calculează probabilitatea de emisie pentru fiecare tag. Variabila cti va căuta tagul în lista cu frecventele unigramului (această fiind descrisă ulterior la secțiunea *Bigram & unigram*), după care se poate aplica formula de la (3.2), unde tf.Value este frecvența de apariție a tagului curent asociat cuvântului din setul de testare, pe (supra) frecvența de apariție a unigramului calculată anterior.

Același proces va fi aplicat listei de cuvinte care încep cu literă mare, doar că rezultatele probabilităților se vor păstra într-o listă diferită (WordCapitalizedTagsEmissionProbabilities).

3.2.3.1.2 Probabilitatea de tranziție

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

$$P(t_i|t_{i-1}) = \frac{c(t_{i-1}, t_i)}{c(t_{i-1})} \tag{3.4}$$

Unde, $P(t_i|t_{i-1})$ – probabilitatea bigramului pentru secvența de taguri t_{i-1}, t_i

$c(t_{i-1}, t_i)$ – frecvența de apariție pentru secvența bigram ale tagurilor t_{i-1}, t_i

$c(t_{i-1})$ – frecvența de apariție a tagului t_{i-1}

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

$$P(VB|NN) = \frac{c(NN, VB)}{c(NN)} = \frac{8027}{13038} \cong 0.61 \tag{3.5}$$

În exemplul demonstrativ de la formula (3.5), se calculează probabilitatea ca „likes” luat ca verb, să urmeze după un substantiv. Numărul de apariții a unui substantiv (NN) urmat de un verb (VB) este 8027, numărul de apariții a unui substantiv (NN) în setul de date este 13038, iar rezultatul împărțirii este egal cu 0.61.

Bigram & unigram

Formula dată la (3.4) este formula de calcul a probabilității unui bigram (sau 2-gram), deoarece aceasta se uită doar la ultimul tag precedent și calculează combinația a 2 taguri unul după altul. Formula care calculează probabilitatea unui singur tag care nu se uită la niciun tag precedent, numită și unigram (1-gram), este următoarea:

$$P(t_i) = \frac{c(t_i)}{N} \tag{3.6}$$

Unde, $P(t_i)$ – probabilitatea unigramului pentru tagul t_i

$c(t_i)$ – frecvența de apariție a tagului t_i

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

Cu formulele (3.2), (3.4) și (3.6) se poate crea un sistem de predicție bazat pe modelul Markov cu stări ascunse de tip bigram.

Implementare Bigram & unigram

Pentru a se putea implementa formulele de la (3.4) și (3.6) mai întâi trebuie obținute frecvențele de apariție a tagului individual (unigram) și a 2 taguri urmate 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 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 (tranziție de tip inițială). 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 precedent

și al doilea parametru fiind tagul curent. Implementarea este foarte asemănătoare la concept cu implementarea de la 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;
   }
}
```

În secțiunea *Emission probability*, a apărut utilizarea acestei metode AddTagToUnigramOccurences(..), în loc de a itera de 2 ori setul de antrenament, se iterează doar o singură dată și se adaugă în același 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 afla câte legături pot exista în total la un n-gram, se dă următoarea formulă:

$$T = x^n \tag{3.7}$$

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

x – numărul tuturor părților de vorbire individuale din setul de date

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),

$$T = x^n = 10^2 = 100 \tag{3.8}$$

Pentru 10 părți de vorbire, un model bigram poate avea în total 100 de legături pentru aceste stări ascunse (ex. …, [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, nu este destul să ne uităm doar la tagul precedent, știind un context mai lung este mult mai util decât să cunoaștem doar cuvântul de dinainte. Pentru asta se introduce trigramul, acesta se uită înapoi la tagul ultimelor 2 cuvinte, față de bigram care se uita doar la tagul precedent. Pentru a calcula probabilitatea trigramului se dă următoarea formulă:

$$P(t_i|t_{i-1}, t_{i-2}) = \frac{c(t_{i-2}, t_{i-1}, t_i)}{c(t_{i-2}, t_{i-1})} \tag{3.9}$$

Unde, $P(t_i|t_{i-1}, t_{i-2})$– probabilitatea trigramului pentru secvența de taguri t_{i-2}, t_{i-1}, t_i

$c(t_{i-2}, t_{i-1}, t_i)$ – frecvența de apariție pentru secvența trigram ale tagurilor t_{i-2}, t_{i-1}, t_i

$c(t_{i-2}, t_{i-1})$ – frecvența de apariție pentru secvența bigram ale tagurilor t_{i-2}, t_{i-1}

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

$$P(NN|JJ, DT) = \frac{c(DT, JJ, NN)}{c(DT, JJ)} = \frac{7303}{11034} \cong 0.66 \tag{3.10}$$

În exemplul anterior, se calculează 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 părților de vorbire pentru un model Markov cu stări ascunse, returnează cea mai bună acuratețe dintre cele trei modele (unigram, bigram, trigram) dar este și cel mai intensiv din punct de vedere al timpului computațional.

Implementare trigram

Acesta va fi implementat tot într-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, în trigram nu se mai salvează și secvențele unde este întâlnit sfârșit de propoziție(sau început de propoziție) la 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 acest lucru ar putea crește performanța sistemului la o acuratețe un pic mai bună (multe propoziții se termină cu un substantiv/verb și încep cu un substantiv sau articol), aceasta nu ar fi utilă în aplicații reale care folosesc un sistem de etichetare a părților de vorbire pentru o propoziție.

Funcția care implementează probabilitatea de la 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);
        }
    }
```

3.2.3.1.3 Netezirea modelelor bigram & trigram (Deleted Interpolation)

Probabilitățile de tip trigram generate dintr-un text corpus *deobicei* nu pot fi folosite direct din cauza valorilor lipsă a secvențelor trigram (sparse-data problem [19]). Pentru a rezolva această problemă, se introduce conceptul de interpolare liniară. Aceasta presupune calcularea unei noi probabilități compuse din suma probabilităților de tranziție (unigram, bigram, trigram) înmulțite cu o pondere determinată:

$$P_{LI}(t_3|t_1,t_2) = \lambda_1 P(t_3) + \lambda_2 P(t_3|t_2) + \lambda_3 P(t_3|t_1,t_2) \tag{3.11}$$

Valorile ponderilor $\lambda_1, \lambda_2, \lambda_3$ sunt estimate prin algoritmul *„Deleted interpolation”*. Pseudocodul general de determinare a acestor ponderi este următorul:

```
set λ1 = λ2 = λ3 = 0
foreach trigram t1, t2, t3 with f(t1, t2, t3) > 0
    depending on the maximum of the following three values:
        case (f(t1,t2,t3)−1)/(f(t1,t2)−1):  increment λ3 by f(t1, t2, t3)
        case (f(t2,t3)−1)/(f(t2)−1):  increment λ2 by f(t1, t2, t3)
        case (f(t3)−1)/(N−1):  increment λ1 by f(t1, t2, t3)
    end
end
normalize λ1, λ2, λ3
```

Figura 3.6 – pseudocodul pentru interpolarea liniară (Deleted Interpolation) pentru trigram[19]

Se poate observa cum probabilitățile fiecărui n-gram sunt scăzute de o constantă = 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 secvențe lipsă dar și pentru a seta ponderile fiecărui n-gram separat. Folosind această funcție, acuratețea predictorului crește, în cazul modelului implementat în această lucrare, creșterea acurateții este mică deoarece tagurile din setul de antrenament sunt deja clasificate în 10 părți de vorbire reprezentative și de acea, dicționarele bigram și trigram au aproape toate permutările posibile. Folosind formula de la (3.7) se poate verifica dacă n-gramul folosit de sistem conține toate permutările sale posibile, astfel trăgând concluzia dacă interpolarea liniară aduce îmbunătățiri majore la predicție.

Înafară de interpolarea realizată pentru trigram, s-a realizat și o interpolare pentru modelul bigram, algoritmul pentru aceasta fiind foarte asemănător cu algoritmul care implementează interpolarea liniară pentru trigram.

Implementarea algoritmului „Deleted Interpolation”

Algoritmul interpolării liniare pentru trigram este următorul:

```
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, această operație de scădere s-a realizat la calcularea finală a probabilității fiecărei tranziții descrisă în subcapitolul anterior. Valorile ponderilor lambda se vor păstra ca membri publici ai clasei Model, aceste valori fiind accesibile mai târziu decodorului pentru a putea aplica formula (3.11).

Algoritmul interpolării liniare pentru bigram este următorul:

```
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 în acest caz nu există 3 ponderi, ci doar 2 ponderi, valorile bigram vor fi adunate la fiecare pondere. Ponderile acestea vor fi și ele accesibile decodorului.

3.2.3.2 Modelul cuvintelor necunoscute

În subcapitolul anterior, am prezentat modelul Markov cu stări ascunse pentru bigram și trigram, folosind doar acest model, sistemul de etichetare poate obține o acuratețe destul de bună dar un sistem foarte bun de etichetare trebuie să poată eticheta corect și cuvintele care nu se găsesc în setul de antrenament. Există mai multe modalități de etichetare a cuvintelor necunoscute precum: sistem bazat pe reguli, învățare nesupravegheată, algoritmi 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 adăugate manual în urma unei analize a setului de date. Funcția finală care va combina aceste 2 componente, va primi 2 parametri de intrare, acestea fiind cuvântul necunoscut și tagul asociat cuvântului. Funcția va returna o probabilitate de asociere a cuvântului cu tagul verificat. Pentru a se putea utiliza funcția ce returnează o probabilitate pentru cuvintele necunoscute, mai întâi vor trebuii obținute probabilitățile de emisie pentru sufixe și prefixe.

3.2.3.2.1 Prefix & suffix – faza de antrenare

Prefixul este un afix pus înaintea rădăcinii unui cuvânt, iar sufixul este pus după rădăcina cuvântului. Cu aceste 2 componente importante se pot obține informații despre partea de vorbire a unui cuvânt.

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 ultimul caracter „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 & calculate din setul de antrenament (timp computațional mare și rezultate mediocre), ci au fost alese manual ca fiind cele mai reprezentative. În urma analizei pe documentele [31], [32], [33], 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 modelul să poată folosi aceste afixuri, trebuie mai întâi să se observe cu ce tag vin asociate în setul de antrenament și să se calculeze probabilitatea de asociere cu tagul întâlnit.

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

$$P_{sp}(x_i|t_i) = \frac{c(t_i, x_i) + \alpha}{\sum_{k=1}^{T_n^{x_i}} k + \alpha d} \tag{3.12}$$

Unde, $P_{sp}(x_i|t_i)$ –probabilitatea de asociere a unui sufix/prefix x_i, cu tagul t_i

$c(t_i, x_i)$ – frecvența de apariție a sufixului/prefixului x_i cu tagul t_i

α – constantă pentru realizarea netezirii „**additive smoothing”** [18]

$\sum_{k=1}^{T_n^{x_i}} k$ – suma tuturor tagurilor asociate sufixului/prefixului x_i

d – mărimea totală a setului de prefixe/sufixe x

Se observă repetiția conceptului de netezire, acesta este foarte important aici deoarece se dorește obținerea unei probabilități mai mari de 0 și pentru sufixele/prefixele care nu sunt întâlnite în setul de antrenament. În sistem, constanta α fost inițializată cu valoarea = 1 , această metodă mai este denumită și „**Laplace Smoothing”**. [18]

ex.

$$P(ly|RB) = \frac{c(RB, ly) + \alpha}{\sum_{k=1}^{T_n^{ly}} k + \alpha d} = \frac{1023 + 1}{1871 + 87} \cong 0.52 \tag{3.13}$$

Aceste calcule prezentate au fost realizate pentru sufixele/prefixele care încep cu literă mică dar în sistem este implementat același concept și pentru calculul sufixelor/prefixelor cuvintelor care încep cu literă mare. Această abordare poate da rezultate mai bune la evaluarea performanțelor sistemului de etichetare.

Implementarea fazei de antrenare a prefixelor & a sufixelor

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

```
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 aici, 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 prefix, cuvântul care începe cu literă mare este convertit la literă mică de metoda .ToLower().

Lista finală cu probabilități va fi una de tip EmissionProbabilisticModel, algoritmul formulei (3.12) pentru cuvinte care încep cu literă 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);
}
```

Același proces este aplicat și cuvintelor care încep cu literă mare.

3.2.3.2.2 Antrenarea modelului pe threaduri separate

În faza de creare a modelului, se crează listele importante cu informații despre probabilitățile de emisie, de tranziție și cele ale sufixelor și ale prefixelor. Rularea neparalelă a funcțiilor care calculează aceste probabilități ar fi destul de ineficientă din punct de vedere al timpului de rulare. Având în vedere faptul că aceste componente sunt separate și nu se influențează una pe cealaltă, acestea pot fi calculate asincron pe diferite threaduri folosind clasa Task din .NET [34]. Implementarea fazei de antrenare a modelului este următoarea:

```
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 este lansat la apelarea funcției .StartNew(). Funcția principală se încheie atunci când toate task-urile au terminat de executat funcția lor proprie.

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();
   }
}
```

3.2.3.2.3 Ponderi bazate pe reguli pentru cuvintele necunoscute

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 în ‘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, aceste 2 valori sunt:

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

Regulile impuse manual sunt implementate astfel:

```
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 cuvântul necunoscut.

3.2.3.2.4 Modelul compus pentru etichetarea cuvintelor necunoscute

Pentru a putea combina aceste componente prezentate anterior, va trebui calculată 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 în următoarea probabilitate finală:

$$P(w_k|t_i) = sp_i + r_i \tag{3.14}$$

Unde, $P(w_k|t_i)$ – probabilitatea cuvântului necunoscut w_j asociat cu tagul t_i

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

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

Formula sp_i se calculează prin a aduna probabilitatea sufixului și a prefixului. Dacă există prefix/sufix pentru cuvântul w_j și tagul curent t_i, probabilitatea de adunare este normalizată cu formula min-max de la (3.15) în intervalul [0, 2]. În cazul când nu este găsit niciun sufix sau prefix, atunci funcția va căuta cea mai mică valoare pentru sufix & prefix în lista cu probabilități de emisie pentru sufixe & prefixe și va continua să execute același proces descris anterior, dar după normalizare, rezultatul obținut va fi înmulțit și cu o constantă egală cu 0.01.

Formula r_i se calculează prin a aduna ponderile pentru regulile care sunt trecute (occurenceAdder adună aceste ponderi), 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 valorea de la variabila worstValueWeight (adică 1.5) este normalizată în intervalul [0, 2.5], 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. Pentru asta, se va executa o funcție care va rotunji valoare la maximul limitei superioare a probabilității (adică 1.0). Depășirea limitei nu este greșită, dacă se trece de limita maximă de 1.0 atunci înseamnă că oricum există o probabilitate de 100% (încredere maximă) ca tagul să fie cel calculat (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 componente care determină tagul cuvintelor necunoscute. Pe lângă asta, formula obține o probabilitate chiar dacă nu se găsesc sufixe/prefixe & nu este trecută nicio condiție pentru cuvântul testat dar poate obține și o probabilitate bună dacă nu se găsesc sufixe/prefixe pentru cuvânt dar în schimb sunt trecute câteva condiții sau invers, dacă sunt găsite sufixe/prefixe dar nu se trece nicio condiție pentru cuvântul testat.

Formula normalizării min-max folosită în sistem este următoarea:

$$z_i = \frac{x_i - \min}{max - \min} \tag{3.15}$$

Unde, $z_i -$ noua valoare

$x_i -$ vechea valoare

min – limita inferioară

max – limita superioară

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

se presupune că nu are niciun sufix sau prefix recunoscut în lista de sufixe/prefixe, iar minimul pentru sufix = 0.10 iar minimul pentru prefix = 0.7 (luate din fișierul cu modelul real cu datele deja antrenate), deci suma probabilității minime a sufixului și a prefixului este = 0.17

Folosind formula de la (3.15) rezultă:

$$sp_i = \frac{x_i - \min}{max - \min} = \frac{0.17 - 0.0}{2.0 - 0.0} = 0.085 \tag{3.16}$$

$$sp_i = sp_i * 0.01 = 0.00085 \tag{3.17}$$

Cuvântul "romana" nu trece nicio condiție impusă anterior, aplicând deci (3.15):

$$r_i = \frac{x_i - \min}{max - \min} = \frac{1.5 - 0.0}{2.5 - 0.0} = 0.6 \tag{3.18}$$

$$r_i = r_i * 0.01 = 0.006 \tag{3.19}$$

Aplicând formula de la (3.14), rezultă:

$$P(w_k|t_i) = sp_i + r_i = 0.00085 + 0.006 = 0.00685 \tag{3.20}$$

Se poate observa cum probabilitatea (3.20) rezultată este foarte mică atunci când nu se trece nicio condiție și nu se găsește niciun sufix/prefix în setul de antrenament.

ex. 2. Cuvântul necunoscut = "Romanian" si tagul = "NN"

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

$$sp_i = \frac{x_i - \min}{max - \min} = \frac{0.239 - 0.0}{2.0 - 0.0} = 0.1195 \tag{3.21}$$

Aici nu se mai înmulțește cu valoarea 0.01, deoarece a fost găsit un sufix în setul de antrenament.

Pentru componenta bazată pe reguli, se trece o singură condiție, cea în care cuvântul începe cu literă mare, deci rezultă:

$$x_i = \frac{2.5}{1.15} = 2.17 \tag{3.22}$$

$$r_i = \frac{x_i - \min}{max - \min} = \frac{2.17 - 0.0}{2.5 - 0.0} = 0.868 \tag{3.23}$$

$$P(w_k|t_i) = sp_i + r_i = 0.1195 + 0.868 = 0.9875 \tag{3.24}$$

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 „Romanian" are tagul de 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 intitulat *Decoder*.

Implementarea modelului compus pentru cuvintele necunoscute

Mai întâi se vor obține probabilitățile sufixelor și a prefixelor pentru cuvântul necunoscut și tagul testat 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 niciun sufix sau niciun prefix pentru cuvântul dat, după care începe căutarea prefixelor/sufixelor pentru cuvântul necunoscut. Se începe prin 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 care nu încep cu literă mare.

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 poate fi mai abstract ca primul găsit. Metoda *ContainsKey*(..)

verifică dacă există cheia cu tagul curent în sufixul/prefixul respectiv, fără a mai trebui iterat și dicționarul de taguri. Algoritmul care determină probabilitatea sufixelelor/prefixelelor pentru cuvintele care nu încep cu literă mare este identic cu cel prezentat anterior.

După finalizarea acestor calcule, se poate trece mai departe la calculul sumei și la normalizare pentru formula sp_i :

```
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 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;
}
```

3.2.4 Blocul Decodor

În subcapitolul anterior am prezentat formarea modelului pentru cuvintele cunoscute, pentru 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) [8].

Un algoritm important de decodificare a unui model ascuns, pe baza programării dinamice, este algoritmul Viterbi. Algoritmul 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 sunt metodele de decodificare bazate pe algoritmul Viterbi. Formula generală de calculare a fiecărui nod la un pas de timp diferit de 0 este următoarea:

$$v_t(j) = max_{i=1}^{N} v_{t-1}(i) a_{ij} b_j(o_t) \tag{3.16}$$

unde, $v_t(j)$ – probabilitatea nodului curent

$v_{t-1}(i)$ – probabilitatea nodului procesat la pasul de timp anterior

a_{ij} – probabilitatea de tranziție de la starea (nodul) anterioară q_i la starea curentă q_j

$b_j(o_t)$ – probabilitatea de emisie (sau state observation likelihood) a observației o_t (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 când există o legătură bigram/trigram între tagul nodului precedent și tagul nodului curent, altfel nu necesită calcularea fiecărui nod de tag deoarece rezultatul ar fi 0.

Un nod Viterbi este implementat astfel:

```
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;
   }
}
```

Acesta se folosește de obiectul PrevNode pentru a putea face backtack la nodul anterior, astfel pentru a putea decodifica întreaga secvență. NextNode este folosit în metoda de backward, iar ambele sunt folosite pentru metoda bidirecțională. Metodele nu calculează formula și pentru noduri unde rezultatul probabilității va fi 0 (nu există probabilitate anterioară/emisie/tranziție).

Funcția care implementează algoritmul 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 setul de testare, al treilea și al patrulea specifică ce tip de tranziție este utilizată (bigram/trigram) și ultimul specifică metoda de decodificare (forward, backward, bidirecțional). Codul subliniat implementează metoda care șterge token-urile (cuvintele) care au tagul de final/început de propoziție.

3.2.4.1 Metoda „Forward”

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 backtrack 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 acea, 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 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.
              GetValueWeightForUnknownWord(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, cuvântul necunoscut se va adăuga în lista de cuvinte necunoscute (se va folosi mai încolo la metricile de evaluare), se va itera lista cu probabilitățile de tranziție bigram și se va căuta ca al doilea tag să fie diferit de tagul de început/sfârșit de propoziție. În momentul când 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 în capitolul anterior), cu tagul curent care va fi salvată în variabila unknownProcent. Valoarea finală este va fi calculată ca produs din probabilitatea de tranziție și probabilitatea cuvântului necunoscut.

După ce această valoare finală 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 precedente a primului cuvânt din propoziție va fi null.

Pentru cazul când se găsește cuvântul/token-ul în lista cu probabilități de emisie atunci se va executa 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 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 această listă în matricea dinamică 2D (listă de listă) numită *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 condiția când s-a ieșit din start, procesul este asemănător, se poate găsi cuvântul în lista cu probabilități de emisie 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) iar parametrul pentru tranziție a fost ales ca trigram, 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 *Deleted Interpolation*. 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. Se poate observa că în condiția subliniată, se vor lua doar tranzițiile unde primul item din tuplul trigram este tagul obiectului cu contextul anterior nodului curent, al doilea item este tagul nodului curent iar al treilea item 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 pentru trigram:

```
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 precedent. Implementarea bigram este foarte asemănătoare cu cea descrisă anterior la trigram, doar că acesta se aplică doar când parametrul de intrare pentru modelul de tranziție este 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 introdus în lista de noduri actuale, la fel ca în procesul descris pentru verificarea primului token din propoziție. La final, matricea 2D cu graful Viterbi este sortată descrescător pentru ca primul nod de pe ultima linie să fie nodul cu probabilitatea cea mai mare atunci când algoritmul de backtracking va avea nevoie de primul nod de pe ultima linie.

Funcția de backtracing care se aplică 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 și contextul tagului în lista de taguri predicționate la fiecare iterație. Ordinea tagurilor rezultate pentru această metodă este diferită față de ordinea de la forward pentru același set de date, afișarea tagurilor din această metodă trebuie evitată pentru a nu emite taguri eronate.

3.2.4.2 Metoda „Backward”

Această metodă presupune parcurgerea propoziției/frazei de la dreapta spre stânga, adică în engleză, de la sfârșit spre început. Această metodă este de cele mai multe ori mult mai bună decât cea forward, ea fiind, defapt, inversul procesului forward. Implementarea acestei metode începe cu iterația de la finalul setului de test spre început ca să poată compara și tagurile în metoda bidirecțională.

Metoda backward folosește 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. Înafară 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 este aproape identic cu cel descris la forward, cu câteva mici modificări. Începutul metodei de backward, realizat 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 backtracking cu metoda backward și la fel și când ajunge la tagul de sfârșit/început de propoziție se poate executa funcția de backtrack.

La implementare, nodul precedent va fi acum cel posterior, iar probabilitățile de tranziție vor fi evaluate de la final spre început (adică de la dreapta spre stânga):

```
......
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). Cu toate că această metodă este foarte asemănătoare cu cea de la 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 funcția 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 metoda backward:

```
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 funcția de backtracking descrisă la **Metoda „Forward”**, diferența aici este că se va itera lista cu istoria nodurilor cu valoarea maximă în loc de matricea 2D a grafului Viterbi, iar backtracking-ul se va realiza de la nodul curent spre nodul posterior (nextNode) până când nodul posterior va deveni null (adică s-a evaluat întreaga propoziție/frază).

3.2.4.3 Metoda Bidirecțională

Această metodă combină ambele metode menționate anterior, am descris că funcția de backtrack va păstra o listă cu istoria nodurilor finale pentru ambele metode. Se cunoaște faptul că, fiecare nod din lista cu istoric reprezintă secvența de propoziție evaluată de metoda forward/backward, atunci se poate compara fiecare nod final din cele 2 liste iar cel cu valoarea cea mai mare va fi ales ca nod pe care se va aplica funcția de backtracking.

Aceasta presupune că valoarea cea mai mare are secvența corectă de taguri deoarece aceasta presupune o încredere mai mare pentru secvența unde nodul final are o probabilitate mai mare. Funcția care implementează metoda bidirecțională 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);
      }
   }
}
```

Pentru cazul când valorile nodurilor sunt egale (foarte rar) atunci cel mai probabil se indică aceleași taguri pentru propoziția respectivă și oricare metodă s-ar alege, rezultatul evaluărilor ar fi exact același.

În figura 3.7, se poate vedea procentul când metoda bidirecțională folosește metoda backward (nodul final este cel mai mare pentru metoda backward), folosește metoda forward și de câte ori valorile maxime ale nodurilor finale pentru metodele forward și backward sunt egale:

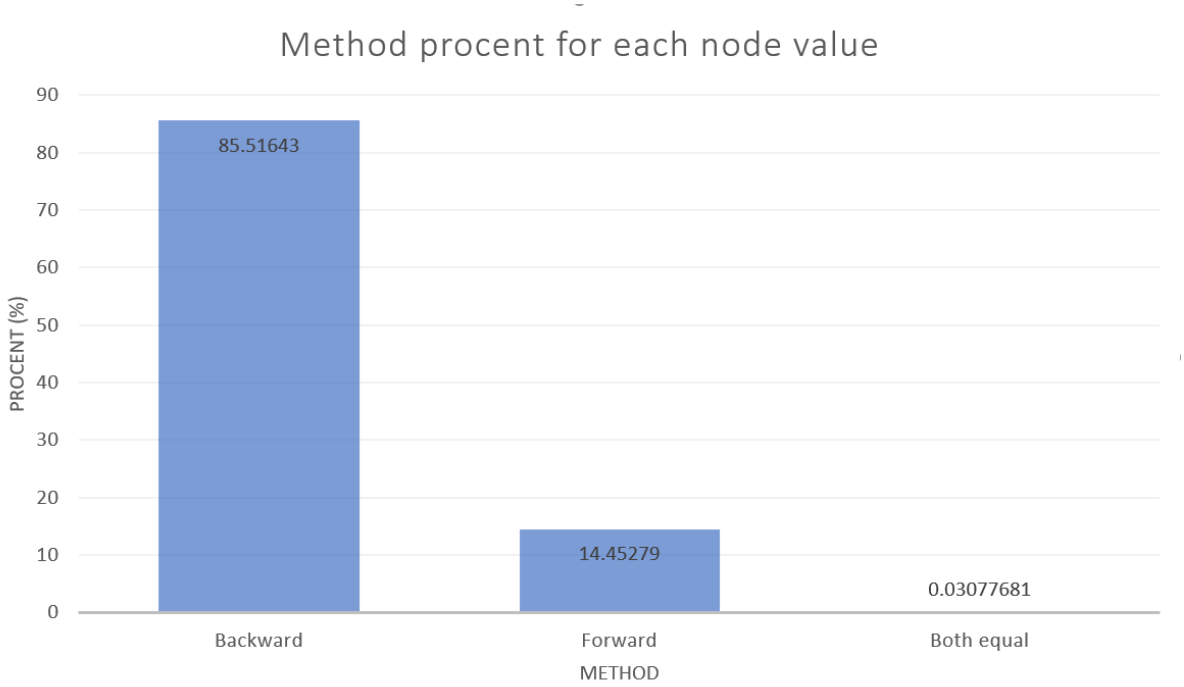

Figura 3.7 – probabilitatea fiecărei metode să fie alese in funcție de valoarea nodului final

3.2.5 Blocul de Evaluare a modelului

După ce decodorul a emis tagurile finale pentru setul de testare, sistemul trebuie să evalueze tagurile acestea pentru a vedea cât de bună este performanța sistemului de etichetare a părților de vorbire. Cum algoritmul acesta este unul de învățare supervizată, setul de testare folosit este etichetat de dinainte cu tagurile reale, cu toate că aceste taguri nu sunt folosite în procesul de învățare și de predicție, ele sunt folosite pentru a evalua performanța algoritmilor de învățare. Evaluarea pentru acest model este împărțită în 2 categorii:

- Evaluarea acurateții simple
- Evaluarea metricilor care folosesc matricea de eroare

3.2.5.1 Acuratețea simplă

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. Aceste metrici pe cuvintele cunoscute/necunoscute sunt foarte utile, deoarece se poate evalua separat acuratețea modelului Markov cu stări ascunse și acuratețea modelului pentru cuvintele necunoscute.

Funcția care implementează acuratețea simplă 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 simplă, 'k' este acuratețea pentru cuvintele cunoscute și 'u' este acuratețea pentru cuvintele necunoscute. Funcția folosește un Set (matematic) de cuvinte necunoscute care a fost creat și încărcat în procesul de decodificare. Acuratețea finală se va calcula ca wordsHit (incrementat dacă tagul la index-ul i pentru setul de testare este același cu tagul predicționat la același index) supra numărul total de cuvinte din setul de testare. Când se alege acuratețea pentru cuvintele cunoscute, metoda verifică dacă la un index oarecare, cuvântul testat se află în setul de cuvinte necunoscute, dacă nu se află atunci acesta continuă cu incrementarea variabilei wordsHit, dacă în caz contrar se află în setul de cuvinte necunoscute, atunci se trece la indexul următor fără să fie procesat și acest cuvânt la indexul respectiv. Pentru cuvintele necunoscute, procesul este asemănător cu cel descris anterior pentru cuvintele cunoscute.

3.2.5.2 Matricea de eroare

Metricile de evaluare care sunt implementate în sistem și folosesc valorile tp,tn,fp,fn din matricea de eroare (în engleză confusion matrix) sunt:

- **Acuratețea (Accuracy)**:

$$A = \frac{tp + tn}{tp + tn + fp + fn} \tag{3.17}$$

- **Precizia (Precision)**:

$$P = \frac{tp}{tp + fp} \tag{3.18}$$

- **Recall-ul (Recall)**:

$$R = \frac{tp}{tp + fn} \tag{3.19}$$

- **Specificitatea (Specificity)**:

$$S = \frac{tn}{tn + fp} \tag{3.20}$$

- **Scorul-F1 (F1-score)**:

$$F_1 = 2\frac{P * R}{P + R} \tag{3.21}$$

Aceste metrici de evaluare menționate anterior 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 (tp, tn, fp, fn), true negative (tn) crește atunci când nici setul de testare și nici tagul predicționat nu sunt aceleași cu tagul curent din iterația setului ClassTags, true positive (tp) crește când și setul de testare și tagul predicționat sunt egale cu tagul curent, false negative (fn) crește când doar setul de testare este același cu tagul curent iar false positive (fp) crește când doar clasa predicționată este aceași cu tagul curent.

După apelarea funcțiilor de evaluare, rezultatele finale pot fi afișate pe ecran pentru a putea fi vizualizată performanța sistemului și acuratețea algoritmilor de învățare implementați în model.

3.3 Rezultate

În acest capitol se vor prezenta rezultatele obținute în urma folosirii acestui sistem de etichetare a părții de vorbire descris în subcapitolele precedente. Evaluarea este făcută pe 6 seturi de parametrii, acestea sunt combinațiile parametrilor modelului Markov cu stări ascunse (bigram/trigram) cu modul/metoda de decodificare (forward/backward/bidirectional). Aceste opțiuni pentru model sunt:

- Forward bigram
- Backward bigram
- Bidirectional bigram
- Forward trigram
- Backward trigram
- Bidirectional trigram.

În continuare, pentru aceste opțiuni menționate anterior, se vor prezenta performanțele metricilor de evaluare care se folosesc de matricea de eroare Tehnica de evaluare este prin metoda 7-30, adică 70% exemple pentru antrenare 30% exemple pentru testare, acestea fiind prezentate doar pentru modelul pe care am obținut cele mai slabe rezultate (forward bigram) și pentru modelul pe care am obținut cele mai bune rezultate (bidirectional trigram). Pe lângă acestea, se vor prezenta și rezultatele acurateței simple pentru cuvintele cunoscute, necunoscute și pentru toate cuvintele, acestea fiind evaluate pe setul de date care folosește metoda crossvalidation pentru împărțirea setului de antrenament și a setului de testare. De asemenea, în afară de acestea se vor prezenta și rezultatele acurateței pentru opțiunile când nu există model iar tagul implicit este substantiv, modelul ”most frequent class baseline” (se alege tagul cu probabilitatea cea mai mare din matricea cu probabilități de emisie) și modelul ”most frequent class baseline” combinat cu tagul implicit de substantiv pentru cuvintele necunoscute.

3.3.1 Performanțele extrase din matricea de eroare & timpul de rulare

Forward bigram:

TAG	ACCURACY (%)	PRECISION (%)	RECALL (%)	SPECIFICITY (%)	F1-SCORE (%)
NN	97.701	94.461	96.445	98.119	95.443
OT	99.856	99.043	99.356	99.905	99.199
CC	99.261	90.929	96.619	99.42	93.688
JJ	98.737	92.343	88.502	99.473	90.381
PP	99.355	97.068	97.887	99.569	97.476
AT/DT	99.309	97.289	97.396	99.594	97.343
VB	98.57	96.408	94.608	99.327	95.499
PN	99.754	99.456	96.798	99.963	98.109
RB	98.811	90.982	86.297	99.517	88.578
TOTAL	99.039	95.331	94.878	99.431	95.079

Figura 3.8 Rezultate obținute de modelul forward bigram

Bidirectional trigram:

TAG	ACCURACY (%)	PRECISION (%)	RECALL (%)	SPECIFICITY (%)	F1-SCORE (%)
NN	97.864	95.699	95.745	98.569	95.722
OT	99.854	99.189	99.186	99.92	99.188
CC	99.369	91.881	97.502	99.481	94.608
JJ	98.826	89.718	93.163	99.233	91.408
PP	99.407	97.737	97.594	99.671	97.665
AT/DT	99.369	98.187	96.938	99.733	97.558
VB	98.637	96.666	94.771	99.376	95.709
PN	99.847	98.962	98.719	99.927	98.841
RB	98.87	90.19	88.471	99.457	89.322
TOTAL	99.115	95.358	95.787	99.485	95.557

Figura 3.9 – Rezultate obținute de pentru modelul bidirectional trigram

Se poate observa că diferența dintre cel mai performant model și cel mai puțin performant model nu este mare, asta datorită setului mare de date. Chiar și așa, indiferent de context, modelul bidirectional trigram este cel mai indicat a se folosi pentru date reale. Toate modele au scorul cel mai mic pentru tagul de adverb și adjectiv, aceste 2 taguri sunt cele mai dependente de context, tagurile dependente de context nu au o probabilitate mare pentru un singur tag și de aceea sunt foarte greu de predicționat în unele contexte. Tagul predicționat cu cel mai bun scor este tagul de Others (alte taguri), acesta conține intejecții, numere (cardinal numbers), cuvinte compuse, etc. care în majoritatea timpului au o formă morfologică unică și nu sunt mereu dependente de context (de exemplu, cuvântul „One” va fi mereu Cardinal number). Rezultatele foarte bune se pot explica prin faptul că setul de date de antrenament și setul de date de test sunt din același corpus de date. Chiar dacă la antrenare sunt alte documente decât la testare acestea provin din aceleași surse (de la aceleași autori) care au tendința să folosească aceleași cuvinte în aceleși contexte și cu aceeași parte de vorbire.

Pentru opțiunea forward bigram, timpul mediu de antrenare a modelului este de 1 minut și 41 de secunde iar timpul mediu de decodificare pentru fiecare secvență este de 1 minut și 44 de secunde (puțin mai mult decât timpul mediu de antrenare). Pentru opțiunea bidirectional trigram, timpul mediu de antrenare a modelului este de 1 minut și 38 de secunde (asemănător cu timpul

mediu de antrenare la bigram-ul forward) iar timpul mediu de decodificare pentru fiecare secvență este de 3 minute și 58 de secunde (aproape 4 minute întregi). Timpul de antrenare între aceste modele nu diferă foarte mult (funcția de antrenare folosește fire paralele pentru a antrena modelul) dar timpul de decodificare este mult mai mare la trigramul bidirecțional deoarece acesta trebuie să calculeze probabiliatea de tranziție trigram și să evalueze modelul atât forward cât și backward, după care să decodifice cea mai bună secvență pentru fiecare propoziție.

Aceste metrici de timp au fost evaluate pe un sistem desktop cu următoarele specificații:

- CPU – AMD Ryzen 5 2500X Quad-Core Processor, cu frecvența de 3.60 GHz
- Installed memory (RAM) – 16.0 GB
- Operating system: Microsoft Windows 10 Pro (64-bit OS)

3.3.2 Acuratețea simplă

Aceasta s-a realizat pe un set de date împărțit prin metode de cross-validation (descrisă în capitolul anterior) cu k = 4, shuffle = true. Rezultatele pentru fiecare model în parte sunt următoarele:

Opțiunea parametrilor	**Procentajul cuvintelor necunoscute (%)**	**Acuratețea pentru cuvintele necunoscute (%)**	**Acuratețea pentru cuvintele cunoscute (%)**	**Acuratețea totală (%)**
Default-tag: NN	13.760	52.351	20.587	24.957
Most frequent class baseline	13.760	0.000	95.851	82.663
Most frequent class baseline + Default-tag: NN	13.760	52.351	95.851	89.866
Forward bigram	3.841	77.373	96.452	95.719
Backward bigram	3.829	82.201	96.507	95.960
Bidirectional bigram	3.830	82.229	96.497	95.951
Forward trigram	3.843	78.280	96.604	95.900
Backward trigram	3.801	81.457	96.630	96.053
Bidirectional trigram	3.832	81.543	96.632	96.054

Figura 3.10 - tabelul de evaluare pentru toate modelele de sistem

Din acest tabel reiese că un model backward trigram este aproape la fel de bun ca cel trigram bidirecțional, această afirmație fiind adevărată deoarece, în peste 85.5% din cazuri se alege backtracking pe branch-ul backward atunci cand se folosește metoda bidirecțională (figura 3.7). Bigramul bidirecțional are cea mai bună acuratețe pentru cuvintele necunoscute dar nu are o acuratețe la fel de bună ca trigramul bidirecțional pentru cuvintele cunoscute. Bigramul forward are o performanță slabă (comparând acest model cu celelalte care utilizează un model Markov), trigramul forward având, bineînțeles, o performanță mai bună față de acesta. Cum 40% de cuvinte din setul de date sunt ambigue, alegand modelul „Most frequent class baseline” + tagul implicit de

substantiv rezultă o acuratețe de doar ~90% de cuvinte predicționate corect [35]. Prezentasem în subcapitolul de preprocesare, cum că substantivul reprezintă aproximativ 23.56% din setul de date, eliminând tagul de sfârșit de propoziție și folosind primul model cu tagul implicit de substantiv, se obține o acuratețe de ~24.95%, ceea ce implică că aproape un sfert din setul de testare a fost ”predicționat” corect. La fel aici ca la metricile prezentate anterior, diferențele între rezultate sunt mici deoarece setul de date este foarte mare.

În figura 3.11, se pot observa creșterea curbelor de învățare pentru modelul trigram bidirecțional, atunci când setul de antrenament primește pe rând, la fiecare pas, o subcategorie din Brown corpus (acestea au fost enumerate și descrise în subcapitolul *Setul de date*):

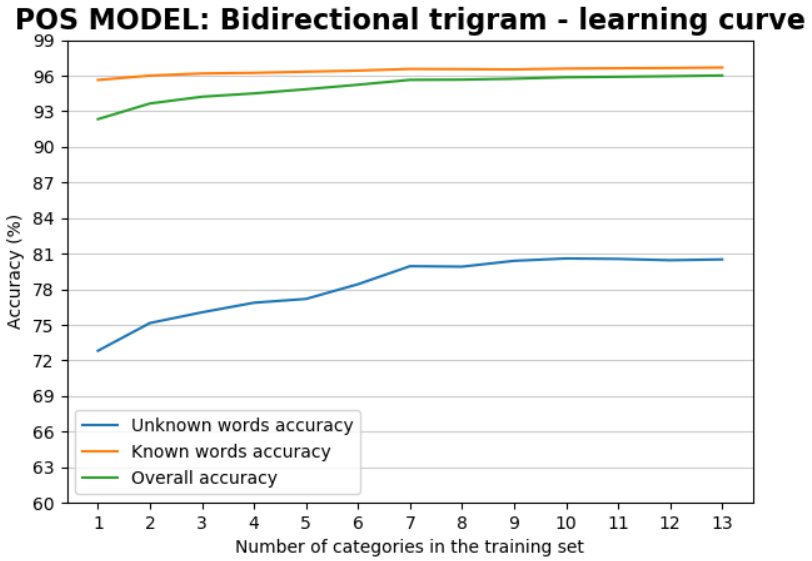

Figura 3.11 – Curbele de învățare pentru modelul Bidirectional trigram

, iar în figura 3.12, se poate observa procentul cuvintelor necunoscute atunci când se adaugă pe rând, la fiecare pas, o subcategorie în setul de antrenare:

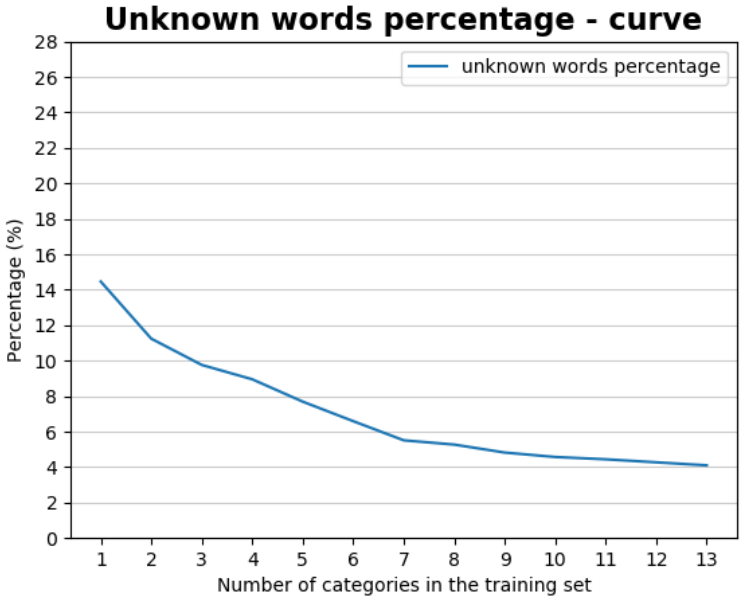

Figura 3.12 – curba pentru procentul cuvintelor necunoscute în modelul Bidirectional trigram

Modelul bidirecțional trigram ajunge la o acuratețe similară cu sistemul de etichetare *TnT* prezentat de Thorsten Brants [19], acesta evaluând performanțele sistemului pe setul de date *Penn Treebank* (diferență de 0.65% între acuratețea totală dintre sistemul prezentat de Brants și sistemul prezentat în această lucrare cu un model trigram bidirecțional).

IV Concluzii

În subcapitolul *Rezultate*, am prezentat rezultatele obținute pentru fiecare parametru al modelului. Modelul cu opțiunea de trigram bidirecțional reușește să obțină cele mai bune performanțe pentru acuratețea totală, specificitatea cea mai mare și f-measure-ul cel mai bun. Acesta, de asemenea, are o fază de antrenare la fel de rapidă ca cea mai simplă opțiune de model (bigram forward), acest timp putând fi îmbunătățit folosind mai puține categorii în faza de antrenare. Dezavantajul major al trigramului bidirecțional este acela că perioada de decodificare durează mult mai mult față de restul modelelor, acest dezavantaj putând fi diminuat prin a returna tagurile după fiecare propoziție în loc de a returna toate tagurile după finalizarea procesului de decodificare. Chiar dacă acest concept nu îmbunătățește timpul în procesul de decodificare, într-o aplicație reală de tip web care ar folosi un sistem de etichetare, acest concept s-ar putea folosi să afișeze tagurile după ce un utilizator introduce o propoziție iar pentru mai multe propoziții introduse, sistemul va procesa fiecare propoziție pe rând și va afișa tagurile după fiecare propoziție procesată. Acest concept este util dar în majoritatea cazurilor nu este și necesar, metrica de timp a fost realizată pe un set de testare care are peste 300.000 de cuvinte.

O aplicație web ce folosește un sistem de etichetare este, de exemplu, *Parts-of-speech.info* [36], aceasta este bazat pe sistemul de etichetare de la *Stanford University*, cu modelul *The Maximum Entropy Markov Model (MEMM)* și algoritmul de decodificare a lui Viterbi bazat pe metoda Greedy de decodificare a unui MEMM [12]. Acest model folosit de Stanford University este un model puțin mai avansat decât modelul Markov cu stări ascunse. O astfel de aplicație web s-ar putea implementa și pentru modelul prezentat în această lucrare, folosind framework-ul ASP.NET pentru aplicații web, partea de backend ar conține partea sistemului de etichetare iar în partea de frontend ar exista un textbox unde utilizatorii ar putea introduce propoziții sau fraze pentru a fi etichetate. De asemenea, modelul prezentat în această lucrare ar putea fi îmbunătățit înainte de fi folosit în aplicații reale, s-ar putea trece de la un model trigram la unul de tip MEMM, s-ar putea folosi rețelele neuronale pentru a învăța forma unui cuvânt necunoscut sau s-ar putea folosi alte procese stohastice pentru a determina partea de vorbire pentru cuvintele necunoscute. Sistemul de etichetare prezentat în această lucrare poate fi folosit și pentru alte scopuri precum: preprocesor pentru o analiză la un nivel mai abstract al datelor în diferite domenii, etichetarea părților de vorbire într-un set foarte mare de date (British National Corpus, Bank of English corpus), indexarea și regăsirea textelor (cuvintele care sunt substantive sau adjective sunt preferabile ca termeni de indexare), procesarea vorbirii și a limbajului [1].

Probleme la crearea unui astfel de sistem de etichetare pot apărea atunci când se lucrează cu un set mare de date (Brown Corpus) iar algoritmii implementați nu sunt bine verificați. Am întâmpinat un minim de probleme în faza de implementare a preprocesării unde tagul rezultat după preprocesare nu era unul valid și alte probleme în faza de implementare a decodorului unde numărul tagurilor rezultate nu era egal cu numărul de cuvinte din setul de testare. Aceste probleme au fost rezolvate prin depanarea pas cu pas a procesului problematic, adaugând astfel și unit-testuri pentru fiecare funcție și algoritm din biblioteca sistemului de etichetare. Problema decodificării a fost rezolvată prin a elimina tagurile de sfârșit de propoziție acolo unde se repetă unul după altul și eliminându-le de tot după procesul de decodificare fără a trebuii să fie și evaluate la final.

În concluzie, sistemul automat de etichetare a părților de vorbire prezentat în această lucrare folosește algoritmi consacrați din domeniul învățării automate și a prelucrării limbajului natural, reușește să obțină performanțe similare cu sisteme de etichetare prezentate în lucrări științifice iar cu câteva modificări minime, acesta poate fi implementat într-o aplicație web sau desktop industrială pentru a lucra pe date reale din limba engleză.

V Bibliografie

[1] Ruslan Mitkov & Atro Voutilainen – The Oxford Handbook of Computational Linguistics (2003)

[2] Parte de vorbire – wikipedia, accesat în data de (12/06/2020): https://ro.wikipedia.org/wiki/Parte_de_vorbire

[3] Artificial Intelligence, articol online, accesat în data de (16/6/2020): https://www.oracle.com/ro/artificial-intelligence/what-is-artificial-intelligence.html

[4] Inteligența artificială – wikipedia, accesat în data de (16/6/2020): https://ro.wikipedia.org/wiki/Inteligen%C8%9B%C4%83_artificial%C4%83

[5] Daniel Volovici – Inteligență artificială și sisteme expert (1997)

[6] Natural language processing – wikipedia, accesat în data de (16/6/2020): https://en.wikipedia.org/wiki/Natural_language_processing

[7] A simple introduction to natural language processing, articol online, accesat în data de (16/6/2020): https://becominghuman.ai/a-simple-introduction-to-natural-language-processing-ea66a1747b32

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

[9] NLP: Part of Speech (PoS) Tagging, articol online, accesat în data de (16/6/2020): https://www.tutorialspoint.com/natural_language_processing/natural_language_processing_part_of_speech_tagging.htm

[10] Part of speech tagging – wikipedia, accesat în data de (16/6/2020): https://en.wikipedia.org/wiki/Part-of-speech_tagging

[11] RDRPOSTagger, documentație online, accesat în data de (16/6/2020): http://rdrpostagger.sourceforge.net/#_Toc435576450

[12] Stanford part of speech tagger – documentație online, accesat în data de (13/06/2020): https://web.stanford.edu/class/cs124/lec/postagging.pdf

[13] Brill tagger - wikipedia, accesat în data de (16/6/2020): https://en.wikipedia.org/wiki/Brill_tagger

[14] Markov chain – wikipedia, accesat în data de (16/6/2020): https://en.wikipedia.org/wiki/Markov_chain

[15] Viterbi Algorithm wikipedia, accesat în data de (15/06/2020): https://en.wikipedia.org/wiki/Viterbi_algorithm

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

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

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

[19] 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

[20] 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

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

[22] 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

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

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

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

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

[27] 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

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

[29] 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

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

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

[32] 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

[33] 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/

[34] 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

[35] Part of speech Tagging, curs 4 (2011-2012), accesat în data de (17/06/2020): https://cs.nyu.edu/courses/spring12/CSCI-GA.2590-001/lecture4.pdf

[36] Parts-of-speech.info – POS tagger online, accesat în data de (13/06/2020): https://parts-of-speech.info/