Universitatea
Transilvania
din Brașov

ȘCOALA DOCTORALĂ INTERDISCIPLINARĂ
Facultatea de Inginerie Electrică și Știința Calculatoarelor

Adrian-Cătălin FLOREA

Sisteme inteligente de decizie

Intelligent Decision Systems

REZUMAT / ABSTRACT

Conducător științific

Prof. dr. mat. Răzvan ANDONIE

BRAȘOV, 2019

Universitatea
Transilvania
din Brașov

COMPONENȚA
Comisiei de doctorat
Numită prin ordinului Rectorului Universității Transilvania din Brașov
Nr. 9892 din 24.05.2019

PREȘEDINTE:	**Conf. dr. ing. Carmen GERIGAN** Decan, Facultatea de Inginerie Electrică și Știința Calculatoarelor Universitatea *Transilvania* din Brașov
CONDUCĂTOR ȘTIINȚIFIC:	**Prof. dr. mat. Răzvan ANDONIE** Universitatea *Transilvania* din Brașov
REFERENȚI:	**Prof. dr. habil. Ioan DZIȚAC** Universitatea *"Aurel Vlaicu"* din Arad
	Prof. dr. ing. Lucian VINȚAN Universitatea *"Lucian Blaga"* din Sibiu
	Prof. dr. ing. Gheorghe TOACȘE Universitatea *Transilvania* din Brașov

Data susținerii publice a tezei de doctorat: 12.07.2019

Eventualele aprecieri sau observații asupra conținutului lucrării vor fi transmise electronic, în timp util, pe adresa acflorea@unitbv.ro.

Totodată, vă invităm să luați parte la ședința publică de susținere a tezei de doctorat.

Vă mulțumim!

Cuprins

		Pag. rez.	Pag. teză
	Prefață	5	6
I	**Noțiuni de bază**	7	8
I.1	**Învățarea automată**	8	9
	I.1.1 Tipuri de algoritmi în învățarea automată	9	10
	I.1.2 Învățarea profundă	12	15
I.2	**Tehnici de învățare automată folosite**	14	18
	I.2.1 Word2Vec, Paragraph2Vec	14	18
	I.2.2 SVM	16	21
	I.2.3 CNN	18	24
	I.2.4 LSTM	19	25
I.3	**Generarea în paralel a numerelor aleatoare**	21	29
I.4	**Tehnici de optimizare a hiperparametrilor**	24	32
I.5	**Apache Spark**	27	35
II	**Atribuirea automată a erorilor în proiectele mari de tip software open source**	30	39
II.1	**Stadiul actual al cercetării**	31	40
II.2	**Implementarea unui sistem de recomandare de tip cluster pentru asignarea automată a erorilor în proiectele mari de tip software open source**	34	44
	II.2.1 Sistemul de recomandare	34	44
	II.2.2 Rezultate și discuții	37	49
	II.2.3 Concluzii	39	59

II.3 Implementarea paralelă a unui sistem de recomandare pentru asignarea automată a erorilor în proiectele mari de tip software open source folosind tehnici de învățare profundă 40 54
II.3.1 Sistemul de recomandare 40 54
II.3.2 Rezultate 42 57
II.3.3 Concluzii 44 59

III Optimizarea hiperparametrilor în învățarea automată 45 60

III.1 Stadiul actual al cercetării 46 61

III.2 Un criteriu dinamic pentru oprirea timpurie în cazul căutării aleatorii 49 64
III.2.1 Algoritmul propus - proprietăți probabilistice 49 64
III.2.2 Experimente 53 73
III.2.3 Concluzii 58 78

III.3 Căutarea aleatorie ponderată 60 79
III.3.1 Metoda WRS 61 79
III.3.2 Aspecte teoretice, convergența algoritmului 62 82
III.3.3 Un exemplu: Optimizarea funcției Grievank 65 85
III.3.4 Optimizarea hiperparametrilor unei rețele de tip CNN 66 88
III.3.5 Concluzii 68 92

IV Concluzii finale, contribuții originale. Diseminarea rezultatelor. Direcții viitoare de cercetare 69 93

IV.1 Concluzii finale, contribuții originale. 70 94
IV.2 Diseminarea rezultatelor. Direcții viitoare de cercetare 73 96

Anexe 82 105

Contents

		Abs. page.	Thesis page
	Preface	5	6
I	**Introductory Notions**	7	8
I.1	**Machine Learning**	8	9
	I.1.1 Machine Learning Algorithms Classification	9	10
	I.1.2 Deep Learning	12	15
I.2	**Utilized Machine Learning Techniques**	14	18
	I.2.1 Word2Vec, Paragraph2Vec	14	18
	I.2.2 SVM	16	21
	I.2.3 CNN	18	24
	I.2.4 LSTM	19	25
I.3	**Parallel Random Number Generators**	21	29
I.4	**Hyperparameters Optimization Techniques**	24	32
I.5	**Apache Spark**	27	35
II	**Automatic Bug Report Assignment in Large Open Source Projects**	30	39
II.1	**State of the Art**	31	40
II.2	**Spark-Based Cluster Implementation of a Bug Report Assignment Recommender System**	34	44
	II.2.1 The Recommender System	34	44
	II.2.2 Results and Discussions	37	49
	II.2.3 Conclusions	39	59

II.3 Parallel Implementation of a Bug Report Assignment Recommender Using Deep Learning 40 54
II.3.1 The Recommender System 40 54
II.3.2 Results 42 57
II.3.3 Conclusions 44 59

III Hyperparameter Optimization in Machine Learning 45 60

III.1 State of the Art 46 61

III.2 A Dynamic Early Stopping Criterion for Random Search in SVM Hyperparameter Optimization 49 64
III.2.1 Proposed Algorithm - Probabilistic Properties 49 64
III.2.2 Experiments 53 73
III.2.3 Conclusions 58 78

III.3 Weighted Random Search for Hyperparameter Optimization 60 79
III.3.1 The WRS Method 61 79
III.3.2 Theoretical Aspects, The Algorithm Convergence 62 82
III.3.3 An Example: Grievank Function Optimization 65 85
III.3.4 Optimizing the Hyperparameters for a CNN Architecture . 66 88
III.3.5 Conclusions 68 92

IV Final Conclusions, Original Contributions Dissemination of Results. Future Research Directions 69 93

IV.1 Final Conclusions, Original Contributions 70 94
IV.2 Dissemination of Results, Future Research Directions 73 96

Appendices 82 105

Prefață

Prezentă teză este structurată în patru părți. Prima parte este o parte introductivă în care sunt prezentate, pe scurt, principalele tehnici și metode folosite. Partea a II-a și a III-a constituie nucleul tezei și prezintă contribuțiile originale. Partea a IV-a prezintă concluziile finale, cu accent pe contribuțiile originale, diseminarea rezultatelor și câteva posibile direcții de cercetare viitoare.

Rezultatele prezentate corespund la două direcții majore de cercetare și anume:

- Realizarea unui sistem de recomandare pentru atribuirea automată a erorilor în proiectele mari de tip software open source. Atribuirea automată a erorilor vine să rezolve o problemă stringentă în contextul în care proiectele de tip software open source devin din ce în ce mai mari atât ca număr de linii de cod cât și ca număr de persoane care intervin în ciclul lor de viață. Atribuirea manuală a erorilor este un proces consumator de timp și din ce în ce mai susceptibil de a genera probleme. În acest context există un interes ridicat pentru automatizarea acestei atribuiri.
- Optimizarea hiperparametrilor în cadrul învățării automate. Optimizarea hiperparametrilor este în prezent una din cele mai explorate direcții din domeniul învățării automate. Atât numărul de parametri pentru algoritmii recent introduși - în special în zona învățării profunde - cât și cantitatea de date disponibile sunt în continuă creștere. Timpul de antrenare necesar și numărul mare de configurații posibile fac ca problema optimizării hiperparametrilor să fie una foarte dificil de rezolvat.

Partea a II-a tratează problema atribuirii automate a erorilor în proiectele mari de tip software open source. Această parte constă din trei capitole. Primul capitol este unul introductiv unde este prezentată problema și este prezentat stadiul actual al cercetării. Capitolul II.2 prezintă o primă propunere pentru un sistem de recomandare în domeniul atribuirii automate a erorilor. Este vorba de o implementare paralelă pe o arhitectură de tip cloud (prima de acest gen, din cunoștințele mele) al cărei nucleu este constituit de un model de tip SVM (Support Vector Machines - Mașini cu Suport Vectorial). Sistemul de recomandare propus obține rezultate similare celor raportate în literatura de specialitate în contextul unei scalabilități net superioare sistemelor desktop existente. Capitolul II.3 prezintă o primă încercare de a folosi tehnici de învățare profundă în zona atribuirii automate

a erorilor. Implementarea propusă este de asemenea una paralelă, scalabilă, bazată pe o arhitectură de tip cloud. Am folosit pentru acest sistem de recomandare, alternativ, CNN (Convolutional Neural Networks - Rețele neurale convoluționale) și LSTM (Long Short-Term Memory - Rețele neurale recurente cu memorie pe termen lung). Rezultatele obținute cu ajutorul CNN sunt la egalitate cu cele mai bune din domeniu și sunt obținute și de această dată, în contextul unei scalabilități mult superioare implementărilor anterioare.

Partea a III-a tratează problema optimizării hiperparametrilor în învățarea automată. Primul capitol este un capitol introductiv care prezintă problema optimizării hiperparametrilor și stadiul actual al cercetării în acest domeniu. Capitolul III.2 propune un criteriu dinamic de oprire automată pentru algoritmul de tip RS (Random Search - Căutare aleatoare) cu aplicare directă în cazul optimizării de hiperparametri pentru modelele de tip învățare automată. Criteriul propus duce la reducerea semnificativă a numărului de încercări necesare algoritmului RS, fără a influența negativ rezultatul obținut în urma optimizării. Este prezentată, de asemenea, o variantă paralelă pentru algoritmul introdus. Probabilitatea de a obține un rezultat optim după un număr redus de încercări crește cu numărul de nuclee folosite de acestă implementare paralelă. Capitolul III.3 introduce o variantă modificată de RS. Algoritmul propus se bazează pe asignarea de probabilități de schimbare pentru fiecare din hiperparametrii țintă. La fiecare iterație, în funcție de aceste probabilități, pentru fiecare hiperparametru, fie este generată o valoare nouă, fie este folosită cea mai bună valoare obținută până atunci. Am demonstrat teoretic că probabilitatea ca algoritmul propus să identifice valoarea optimă căutata, după un număr de încercări dat, este mai mare decât în cazul RS. Am testat algoritmul în contextul optimizării unei variații a funcției Grievank cât și pentru optimizarea hiperparametrilor unei rețele neurale de tip CNN. În ambele cazuri rezultatele obținute au fost superioare celor obținute de alte tehnici de optimizare.

Partea I

Noțiuni de bază

Aceasta parte reprezintă o parte introductivă, în care sunt prezentate pe scurt principalele tehnici și metode folosite în cadrul tezei. Primul capitol prezintă noțiuni introductive legate de învățarea automată. Ulterior sunt evidențiate tehnicile de învățare automată folosite în partea a II-a și a III-a. Prezentarea continuă cu câte un capitol dedicat generării în paralel a numerelor aleatoare și, respectiv, tehnicilor de optimizare a hiperparmetrilor folosite pe parcursul tezei. Ultimul capitol din partea curentă descrie colecția de biblioteci software Apache Spark.

Capitolul I.1
Învățarea automată

Acest capitol prezintă concepte și noțiuni de bază din zona învățării automate cum ar fi învățarea supervizată, nesupervizată și prin recompensă. Sunt prezentate, de asemenea, noțiuni generale legate de învățarea profundă.

Sintagma *învățare automată* (Machine Learning - ML) se referă la detectarea în mod automat a șabloanelor semnificative din date. Învățarea automată este unul din domeniile cu cea mai rapidă creștere din zona științei calculatoarelor. În ultimii ani tehnicile specifice acestui domeniu au devenit o unealtă standard în rezolvarea problemelor care constau în extragerea de informație din seturi mari de date. Recent, zona învățării automate a luat amploare atât din perspectiva aplicațiilor dezvoltate cât și prin prisma noilor descoperiri în zona teoretică a domeniului. Algoritmi din zona învățării automate sunt din ce în ce mai prezenți în aplicațiile pe care la folosim în mod curent, cum ar fi motoarele de căutare online [67], aplicațiile de tip anti-spam [54], securizarea tranzacțiilor cu card de credit [16] etc. dar și în domenii de nișă precum mașini autonome [13], securitate cibernetică [14], medicină [46].

În linii mari, învățarea automată constă în programarea calculatoarelor pentru a optimiza un criteriu de performanță folosind exemple de date sau experiența anterioară [3]. Învățarea automată este necesară atunci când nu poate fi scris un program care rezolvă direct o problemă dată. Un alt caz când o abordare de tip învățare automată este necesară apare atunci când problema țintă se schimbă în timp sau depinde de un anumit mediu exterior.

Pentru a rezolva o problemă cu ajutorul tehnicii de calcul este necesar un algoritm, o secvență de instrucțiuni care trebuie executate pentru a transforma datele de intrare în rezultat [3]. Pentru anumite categorii de probleme, însă, acest algoritm sau nu există sau nu este încă descoperit. Tot ce este disponibil este o mulțime de date de intrare (I) și, eventual, rezultatul dorit pentru fiecare din datele din mulțimea respectivă (O) dar nu și funcția matematică (F) care transformă intrările în ieșiri. În acest caz, ce se poate face este instruirea calculatorului să ”învețe” funcția matematică $F : I \to O$. În cele mai multe cazuri o astfel de funcție nu poate fi exact identificată dar poate fi construită o aproximare suficient de bună, conform cu un

criteriu de performanță stabilit. Ținta unui algoritm de tip învățare automată va fi, așadar, optimizarea acestui criteriu de performanță definit, plecând de la datele de intrare furnizate sau de la experiența anterioară.

Sistemele inteligente de decizie folosesc cunoștințe de statistică în crearea modelelor matematice și necesită algoritmi eficienți pentru rezolvarea problemei de optimizare descrise cât și capacitatea de a stoca și procesa volume de date în continuă creștere. Odată obținut un model, reprezentarea acestuia precum și modul în care va fi folosit trebuie, de asemenea, sa fie eficiente. Pentru anumite aplicații, eficiența algoritmului de optimizare poate fi la fel de importantă ca performațele modelului obținut (în special în cazul în care volumul de date disponibil este mare sau în continuă creștere). Exemplele de sisteme inteligente de decizie propuse în prezenta teză se concentrează atât pe performanța modelelor obținute cât și pe scalabilitatea algoritmilor propuși și abilitatea lor de a lucra cu volume de date mari sau din ce în ce mai mari.

I.1.1 Tipuri de algoritmi în învățarea automată

Din perspectiva tipului de date disponibile și a tipului de rezultat căutat (predictiv, descriptiv etc.) algoritmii de tip învățare automată pot fi clasificați în mai multe categorii. Secțiunea curentă prezintă, în linii mari, aceasta clasificare.

I.1.1.1 Învățarea supervizată

Regresie

Plecând de la un set de date de intrare **X** și o valoare numerică $r_j, j = 1, \ldots, n$ asociată fiecărui element $X_j, j = 1, \ldots, n$ din acest set, problema regresiei constă în determinarea valorii r^* pentru un element nou, X^*, element pentru care această valoare nu este cunoscută. Scopul algoritmului de regresie este așadar identificarea unei funcții $F^* : \mathbf{X} \rightarrow \mathbb{R}$, care aproximează, în limitele unei erori acceptabile, transformarea lui X_j în $r_j, j = 1, \ldots, n$. Intuitiv, acestă funcție va fi capabilă să prezică valorile corespunzătoare unor elemente noi.

Elementele setului de date poată numele de observații. Fiecare observație este descrisă de anumite caracteristici (features). Aceste caracteristici pot fi numerice sau categoriale (indicatori ai apartenenței la o anumită categorie). La rândul lor, caracteristicile numerice sunt fie discrete (de exemplu numărul de cuvinte într-un email) fie continue (cantitatea de precipitații pentru o anumită suprafață și interval

de timp). Orice alt tip de informație prezentă în datele de intrare va fi convertită la unul din aceste tipuri de caracteristici. De exemplu, informația textuală este transformată în informație numerică folosind tehnici ca TF/IDF [71] sau Word2Vec [56].

Matematic, rezultatul unui algoritm de regresie polinomială, aplicat pe un set de observații **X**, unde fiecare observație are d caracteristici, este o funcție de forma:

$$y = \sum_{i=1}^{m} W_i \cdot X^i + b \tag{I.1.1}$$

unde m este gradul polinomului folosit pentru aproximare, b este un scalar, W_i este un vector de dimensiune $(1, d)$ iar X^i este un vector de dimensiune $(d, 1)$ ale cărui elemente sunt ridicate la puterea i. $W_i \cdot X^i$ reprezintă înmulțirea scalară a celor doi vectori. Cu cât valoarea lui m crește, cu atât aproximarea valorilor din setul de date de intrare este mai exactă. Nu întotdeauna însă, un model care aproximează foarte precis datele de antrenare este și modelul care generalizează cel mai bine [3], în mare parte din cauza zgomotului care poate interveni în datele de antrenare disponibile.

Clasificare

Plecând de la un set de date de intrare și apartenența fiecărui element al acestui set la una sau mai multe clase, problema clasificării constă în identificarea clasei (claselor) căreia îi aparține un element nou, element pentru care această asociere nu este cunoscută. Clasificarea poate fi astfel privită ca o particularizare a regresiei, pentru care codomeniul este o mulțime discretă, finită și fiecare element al codomeniului reprezintă o anumită clasă. Cel mai simplu exemplu de clasificare este clasificarea binară, atunci când sunt considerate doar două clase. Cazurile mai complicate constau în seturi de date unde mai multe clase sunt considerate (clasificare multi-clasă). Pentru aceste seturi de date, fiecare element poate aparține uneia (clasificare single-label) sau mai multor clase (clasificare multi-label).

Apartenența fiecărei observații la una sau mai multe din clasele țintă poate fi, de asemenea, modelată sub forma unui vector r_j, $r_{j,k} = I(j, k)$ unde:

$$I(j,k) = \begin{cases} 1, \text{ dacă } X_j \text{ aparține clasei } k \\ 0, \text{ altfel} \end{cases} \tag{I.1.2}$$

Problema clasificării constă astfel în determinarea vectorului r^* corespunzător lui X^*, un punct care nu face parte din setul de date de intrare (denumit și set de date de antrenare).

I.1.1.2 Învățarea nesupervizată

În învățarea supervizată, scopul este determinarea unei corespondențe între datele de intrare și cele de ieșire. Valorile datelor de ieșire sunt furnizate de către un "supraveghetor". În cazul învățării nesupervizate, acest supraveghetor nu există, tot ce este disponibil sistemului sunt datele de intrare. În cadrul acestor date este presupusă o anumită structură, anumite șabloane apar mai des decât altele, iar scopul învățării nesupervizate este tocmai identificarea acestor șabloane.

O metodă comună de estimare a densității este gruparea (clustering). Scopul acestei metode este identificarea de grupuri (clustere) în datele de intrare. Identificarea acestor grupuri se face pe baza unei măsuri de similaritate între instanțele din setul de date de intrare. Pentru a putea calcula similaritatea dintre două documente, acestea sunt reprezentate sub forma unor mulțimi de cuvinte ("bag of words" - BOW). Acestă reprezentare constă în definirea unui dicționar de N termeni și reprezentarea fiecărui document sub forma unui vector N-dimensional binar. Elementul de pe poziția i este unu dacă termenul de indice i apare în document și zero în caz contrar. Pentru a putea limita numărul de cuvinte din dicționar, de cele mai multe ori cuvintele derivate sunt reduse la rădăcina lor comună (infinitivul pentru verbe, nominativ singular pentru substantive etc.) iar cuvintele foarte comune (prepoziții, articole) sunt ignorate. Documentele sunt apoi grupate în funcție de numărul de cuvinte comune.

Același gen de abordare funcționează indiferent de natura datelor de intrare (ADN sau lanțuri de proteine în bioinformatică, secvențe video sau audio, imagini etc.). Atât timp cât o reprezentare numerică a acestor date de intrare poate fi construită atunci poate fi calculată similaritatea dintre oricare două instante și implicit datele pot fi grupate.

I.1.1.3 Învățarea prin recompensă

În cazul anumitor sisteme de optimizare, ieșirea este reprezentată de o secvență de acțiuni. În acest caz, o singură acțiune în sine nu este importantă, ce este important este secvența corectă de acțiuni care duce la îndeplinirea unui scop. În oricare din stările intermediare ale sistemului nu există o acțiune optimă, o acțiune este

considerată bună cât timp face parte dintr-o secvență de acțiuni bună. În cazul unui astfel de sistem, un algoritm de învățare automată trebuie să fie capabil să evalueze rezultatul unei secvențe de acțiuni și să învețe din secvențele anterioare pentru a genera o nouă secvență.

Cel mai comun exemplu pentru un astfel de sistem este un joc (șah, GO etc.) unde o singură mutare nu este importantă în sinea ei, ce contează este secvența de mutări care, potențial, duce la câștigarea jocului.

Învățarea prim recompensă (reinforcement learning) rezolvă problema dificilă a corelării acțiunilor imediate cu o recompensă întârziată. Un sistem de învățare prin recompensă este caracterizat de:

- Unul sau mai mulți agenți - algoritmul (algoritmii) care este responsabil pentru luarea deciziilor
- Un set de acțiuni posibile - la orice moment algoritmul poate alege una din aceste acțiuni
- Starea sistemului - situația în care se află agentul (agenții)
- Recompensa - constituie feedback-ul prin care este înregistrat succesul sau eșecul unei acțiuni efectuate de agent
- Politica - este strategia conform căreia agentul decide următoare acțiune în funcție de starea curentă a sistemului

I.1.2 Învățarea profundă

Învățarea profundă (Deep Learning - DL) își propune să rezolve problemă învățării automate prin introducerea unor modele - reprezentări ale realității - compuse din alte modele din ce în ce mai simple. Învățarea profundă permite computerelor să construiască concepte complexe plecând de la o ierarhie de alte concepte similare.

Conform cu Goodfellow *et al.* [32], exemplul central de tehnică specifică învățării profunde este o rețea de tip perceptron multistrat (Multilayer perceptron - MLP [74]). Acest MLP este, în esență, doar o funcție matematică care transformă valorile de intrare în valori de ieșire.

Există două modalități de a măsura profunzimea unui model. Prima se bazează pe numărul de instrucțiuni secvențiale care trebuie executate pentru a evalua arhitectura, echivalent cu lungimea celei mai lungi căi care permite transformarea unor date de intrare în date de ieșire. Cea de-a doua modalitate se bazează pe profunzimea grafului care descrie relațiile dintre concepte. Din această perspectivă,

de exemplu, un algoritm care detectează o față umană plecând de la detecția unui ochi folosește două concepte și are profunzimea doi. Din prisma instrucțiunilor secvențiale executate, profunzimea este însă $2n$, unde n este numărul de pași necesar pentru identificarea unui concept (ochi sau față în cazul exemplului dat). Deoarece nu este întotdeauna clar care din aceste două abordări trebuie folosită nu există o singură valoare corectă pentru profunzimea unei arhitecturi [32]. Similar, nu există un consens cu privire la cât de profund trebuie să fie un model pentru a fi considerat că aparține zonei de învățare profundă. În lipsa unei limite bine definite, învățarea profundă poate fi așadar privită ca domeniul care implică modele fie cu un graf computațional, fie cu un graf conceptual, mai adânc decât modelele tradiționale din învățarea automată.

O definiție general acceptată [66] prezintă învățarea profundă ca fiind clasa de modele din învățarea automată care:

- Folosesc o succesiune de mai multe straturi de transformări neliniare. Fiecare strat folosește ca date de intrare datele de ieșire din stratul precedent.
- Pot fi aplicate atât pentru învățare supervizată cât și nesupervizată.
- Învață mai multe niveluri de reprezentare care corespund unor niveluri diferite de abstractizare. Aceste niveluri formează un concept ierarhic.

Capitolul I.2
Tehnici de învățare automată folosite

Acest capitol prezintă, în mare, tehnicile importante din domeniul învățării automate folosite pe parcursul acestei teze. Capitolul debutează cu descrierea tehnicilor de tip Word2Vec și Paragraph2Vec [49], tehnici folosite în Capitolul II.3. Capitolul continuă cu prezentarea generală a algoritmului de tip Support Vector Machines (SVM) [21], algoritm folosit în capitolele II.2 și III.2. Capitolul se încheie cu prezentarea tehnicilor de învățare profundă: CNN [50] și LSTM [39], tehnici folosite în capitolele II.3 și III.3.

I.2.1 Word2Vec, Paragraph2Vec

Word2Vec [55] și Paragraph2Vec [49] fac parte din familia algoritmilor bazați pe vectori de încorporare (embedding vectors). Acest tip de algoritmi propun o metodă de rezolvare a problemelor întâmpinate de cea mai populară modalitate de a transforma informația textuală în reprezentarea ei numerică și anume mulțimile de cuvinte (Bag of Words - BOW). Tehnica BOW are la bază construirea unui dicționar cu N termeni de interes (de obicei N este foarte mare) și apoi, transformarea textului de intrare într-un vector binar, N-dimensional, strict pe baza prezenței fiecărui termen din dicționar în text. Principalele probleme ale unei astfel de abordări sunt cauzate de pierderea informației referitoare la ordinea cuvintelor în text cât și a sensului fiecărui cuvânt.

Word2Vec

Algoritmii de tip word embeddings se folosesc de contextul lingvistic în care fiecare cuvânt este prezent pentru a construi reprezentarea lui numerică. Word2Vec, introdus de Mikolov *et al.* [55], folosește ca date de intrare un corpus și produce un spațiu vectorial, în general cu câteva sute de dimensiuni, unde fiecărui cuvânt din corpus îi este atribuit un vector din acest spațiu. Spațiul este astfel construit încât vectorii corespunzători cuvintelor care apar într-un context similar sunt situați la distanțe mici intre ei. Cea mai simplă arhitectură de tip Word2Vec se bazează

pe o rețea neurală cu două straturi care este folosită pentru a reconstrui contextul lingvistic al cuvintelor. Atât vectorii de intrare cât și cei de ieșire sunt vectori binari pentru care o singură valoare este unu și restul sunt zero. În cazul acestora valoarea unu se află pe poziția corespunzătoare cuvântului cu cea mai mare probabilitatea de a apărea în text în condițiile în care a fost observat cuvântul reprezentat de vectorul de intrare.

Pentru a putea prezice un cuvânt țintă plecând de la un context mai larg, există două variații importante ale algoritmului Word2Vec în funcție de modul în care sunt folosite date de intrare:

- **C-BOW** - această tehnică presupune folosirea mai multor vectori de intrare (vectori context) pentru a determina dependența cuvântului țintă de mai multe cuvinte de intrare. Pentru un context de dimensiune C, stratul de intrare în rețea este replicat de C ori iar fiecare neuron din stratul ascuns efectuează o operație de împărțire la C - valorile obținute reprezintă astfel media valorilor corespunzătoare fiecărui cuvânt din context.
- **Skip-Gram** - acestă arhitectură este inversul modelului C-BOW. Intrarea este reprezentată de cuvântul țintă iar ieșirea constă în cuvintele context. Stratul ascuns este definit la fel ca pentru arhitectura de bază dar stratul de ieșire calculează C distribuții pentru un context C dimensional.

Algoritmul de antrenare pentru cele trei arhitecturi este prezentat în [73]. Fiecare din cele două variații are propriile avantaje și dezavantaje. Skip-Gram se comportă mai bine pentru seturi de date mici și reprezintă mai eficient cuvintele rare iar C-BOW are un timp de antrenare mai scăzut și o reprezentare mai bună pentru cuvintele frecvente [56].

Paragraph2Vec

Paragraph2Vec este introdus de Le și Mikolov [49] ca o extensie a Word2Vec. Word2Vec transformă un singur cuvânt într-un vector d dimensional. De cele mai multe ori însă datele de intrare pentru un algoritm de tip învățare automată sunt reprezentate de texte cu mai mult de un cuvânt. Este necesară așadar o reprezentare numerică a unei secvențe de text compusă din mai multe cuvinte. Cea mai simplă reprezentare de acest gen este o medie ponderată a vectorilor corespunzători fiecărui cuvânt din secventă. Acestă abordare suferă de aceeași problemă cu BOW deoarece pierde informația legată de ordinea cuvintelor. Există și alte moduri mai elaborate de a combina reprezentările vectoriale ale cuvintelor dar limitate la o singură propoziție [49].

Paragraph2Vec este capabil să construiască reprezentări pentru text de lungime variabilă. Algoritmul folosește ca date de intrare, pe lângă reprezentarea ca vector binar a cuvintelor din context, un alt vector care reprezintă paragraful. Acest vector este concatenat la vectorii tuturor cuvintelor. Valorile vectorului sunt partajate pentru toate cuvintele aparținând aceluiași paragraf dar nu și între paragrafe diferite. Acest vector funcționează astfel ca un identificator pentru paragraf. După etapa de antrenare vectorul obținut pentru fiecare paragraf poate fi folosit ca reprezentare a acestuia.

I.2.2 SVM

SVM (Support Vector Machines - Mașini cu Suport Vectorial) este un algoritm de clasificare binară introdus de Vapnik în [84]. Pentru un set de date de antrenare și două clase de ieșire, SVM construiește un model care atribuie o parte din date unei clase și restul, celeilalte clase. Toate instanțele din setul de date de antrenare sunt reprezentate ca vectori într-un spațiu d-dimensional iar SVM încearcă să clasifice acești vectori. În spațiul d-dimensional, hiperplanul care separă cele două clase are ecuația:

$$y = W \cdot X + b \tag{I.2.1}$$

unde W (vector de dimensiune $(1, d)$) determină orientarea hiperplanului și termenul liber b (scalar) este deplasarea (bias) care decide poziția hiperplanului în spațiul d-dimensional. În context SVM, aceste hiperplane sunt de fapt suprafețe liniare de separare și nu este necesar ca ele să treacă prin origine. Ecuația (I.2.1) poată numele de funcție de discriminare liniară. Acestă funcție va avea o valoare pozitivă pentru vectorii dintr-o anumită parte a hiperplanului și o valoare negativă pentru vectorii din cealaltă parte a hiperplanului.

Între instanțele a două clase separabile pot exista mai multe hiperplane de separare. SVM încearcă să minimizeze șansele ca oricare din datele de intrare să fie clasificată incorect. Intuitiv, aceasta este echivalent cu maximizarea marginii dintre hiperplanul de separare și instanțele oricărei din cele două clase.

Dacă considerăm **X** setul de date de antrenare și cele două clase ca fiind $y_1 = 1$, pentru $W \cdot X + b > 0$ și $y_2 = -1$, pentru $W \cdot X + b < 0$ atunci avem:

$$y_i(W \cdot X_i + b) \geq 0, \forall X_i \in \mathbf{X} \tag{I.2.2}$$

iar pentru a asigura robustețea clasificării (I.2.2) devine, pentru un set de date normalizat:

$$y_i(W \cdot X_i + b) \geq 1, \forall X_i \in \mathbf{X} \tag{I.2.3}$$

Dacă $W \cdot X_i + b = 1$ sau $W \cdot X_i + b = -1$ atunci X_i poartă numele de vector suport - de aici și denumirea algoritmului: SVM = mașini cu suport vectorial. Dacă ignorăm oricare din punctele(vectori) care nu satisfac egalitatea precedentă, hiperplanele de separare nu se schimbă, dacă, în schimb, oricare din vectorii pentru care acestă egalitate este verificată sunt modificați atunci și hiperplanul de separare se va modifica. De aceea acești vectori se consideră că oferă *suport* hiperplanului de separare.

Scopul SVM este acela de a maximiza distanța dintre hiperplanele date de $W \cdot X + b = 1$ și $W \cdot X + b = -1$ cu constrângerea că niciun punct din setul de antrenare nu trebuie să se situeze între cele două hiperplane.

Pentru cazul în care clasele nu sunt complet separabile sau când sunt acceptate valori atipice (outliers), condiția (I.2.3) poate fi scrisă sub forma relației (I.2.4).

$$y_i(W \cdot X_i + b) \geq 1 - \xi_i, \xi_i \geq 0, \forall X_i \in \mathbf{X} \tag{I.2.4}$$

SVM se transformă astfel într-un clasificator *soft*, dispus să accepte și erori de clasificare.

Cazul liniar neseparabil, extensia SVM pentru mai multe clase

Pentru cazul liniar neseparabil, vectorii de intrare sunt transformați cu ajutorul unei funcții $\Phi(X_i), X_i \in \mathbf{X}$ din spațiul inițial într-un alt spațiu în care șansele ca ei sa devină liniar separabili cresc. Algoritmul SVM folosește apoi ca date de intrare vectorii din spațiul rezultat. Algoritmul rămâne același, X_i fiind înlocuit cu $\Phi(X_i)$. Algoritmul SVM este astfel definit încât calculele necesare depind exclusiv de calculul produsului a două elemente, $< X_i, X_j >$ și nu de valorile efective pentru X_i și X_j. Plecând de la această observație rezultă că pentru a aplica SVM pe spațiul obținut în urma aplicării transformării Φ sunt necesare exclusiv valorile produselor $< \Phi(X_i), \Phi(X_j) >$ și nu calculul efectiv a lui Φ în fiecare punct. Acest fapt permite aplicarea unor funcții de transformare ale căror valori pot fi foarte dificil de calculat, cu condiția ca produsul $< \Phi(X_i), \Phi(X_j) >$ să poată fi calculat ușor.

Produsul dat de $< \Phi(X_i), \Phi(X_j) >$ se numește nucleu (Kernel) și există câteva variante standard de nuclee care se folosesc în cazul SVM. Cele mai des întâlnite sunt:

- Liniar $K(X_i, X_j) = \langle X_i \cdot X_j \rangle$
- Polinomial $K(X_i, X_j) = (\alpha \langle X_i \cdot X_j \rangle + c)^d$ unde α este panta, c este o constantă și d este gradul polinomului
- Gaussian $K(X_i, X_j) = e^{-\gamma ||X_i - X_j||^2}$

Aplicarea unei funcții de tip nucleu permite transformarea datelor într-un spațiu în care acestea devin potențial separabile și clasificarea lor în acest spațiu.

Deși algoritmul SVM clasic este construit pentru două clase, el poate fi extins și la clasificarea multi-clasă folosind tehnici de tip "unu la unu - OnevsOne" sau "Unul împotriva tuturor - OnevsAll". Tehnica OnevsOne constă în construirea de clasificatoare pentru fiecare pereche de clase, clasa pentru un vector nou determinându-se prin vot (pentru fiecare exemplu se calculează clasa corespunzătoare și se alege clasa cu cele mai multe rezultate obținute). OnevsAll constă în antrenarea unui model pentru fiecare din clasele țintă, celelalte clase fiind grupate. În final se alege clasa care are valoarea maximă pentru funcția învățata.

I.2.3 CNN

Rețelele Neurale Convoluționale (Convolutional Neural Networks - CNN) reprezintă o categorie de rețele neurale profunde care se dovedesc foarte eficiente în domeniul recunoașterii și clasificării de imagini. Modul în care este construită arhitectura acestor rețele le face să nu necesite identificarea prealabilă a caracteristicilor semnificative din datele de intrare, rețelele învățând filtrele care altfel ar fi trebuit definite apriori. Această independență de intervenția umană în pre-procesarea datelor constituie principalul avantaj al CNN în raport cu tehnici alternative.

O rețea convoluțională constă într-un strat de intrare, unul de ieșire și o secvență de straturi ascunse. În mod obișnuit straturile ascunse ale unui CNN sunt straturi convoluționale, straturi de activare (de obicei ReLU - $f(x) = max(0, x)$), straturi de tip pooling, dense sau straturi de normalizare.

Straturile de tip convoluțional constituie nucleul rețelelor de tip CNN. Parametrii unui astfel de strat constau într-un set de filtre pe care rețeaua le învață. Un astfel de filtru este caracterizat prin trei dimensiuni: lungime, lățime și adâncime. Lungimea și lățimea sunt, în general, valori mici iar adâncimea este întotdeauna egală cu adâncimea datelor de intrare - în cazul unei imagini color, de exemplu, adâncimea

este trei și este dată de numărul de canale de culoare. În timpul fazei de antrenare fiecare filtru este glisat cu un anumit pas pe întreaga suprafață a datelor de intrare și se calculează produsul scalar dintre valorile filtrului și datele de intrare la fiecare poziție. Toate straturile convoluționale sunt urmate de o funcție de activare (în general ReLU). Staturile de tip pooling determină o grupare a datelor lor de intrare. Rolul acestora este de a reduce periodic dimensiunea datelor pentru a reduce astfel numărul de parametri și efortul computațional necesar în antrenarea rețelei. Un strat de tip pooling operează independent pe fiecare nivel de adâncime al datelor de intrare și îl redimensionează folosind în general o operație de tip MAX (maxim) sau AVG (medie). Straturile dense sunt aceleași din rețelele neurale clasice, acestea au conexiuni către toți neuronii din stratul anterior. În mare, pentru o rețea CNN straturile de tip convoluțional și pooling extrag caracteristici ale datelor de intrare iar straturile dense ajută la clasificarea acestora.

I.2.4 LSTM

Rețelele neurale tradiționale iau în calcul, la fiecare iterație, exclusiv starea curentă a datelor de intrare. Ele ignoră astfel, deliberat, contextul general și modul în care informația a evoluat până la starea ei actuală. Rețele recurente (Recurrent Neural Networks - RNN) propun o modalitate de a integra informații despre stările precedente ale sistemului în modul de calcul al stării curente, similar cu modul în care un om, de exemplu, ține cont de paragraful precedent în momentul în care citește o secvență de text.

Rețelele de tip LSTM, introduse de Hochreiter și Schmidhuber [39], reprezintă o variație a rețelelor RNN. Autorii au demonstrat că acestea funcționează, pentru un număr mare de cazuri, mai bine decât rețelele RNN clasice. Uneori, execuția cu succes a unei sarcini depinde de informația recentă. Un astfel de exemplu este un model care prezice care este următorul cuvânt într-un text, plecând de la cuvintele anterioare. În cazul acestor modele nu este necesar un context mai general, informația relevantă și cea necesară sunt apropiate. Există însă și situații când contextul necesar pentru a identifica informația relevantă este mult mai larg. Pentru acest caz, rețelele LSTM devin net superioare celor RNN clasice, emulând un fel de memorie de lungă durată (Long-Term Memory) pe lângă memoria de scurtă durată (Short-Term Memory). De aici și numele acestor rețele: Long Short-Term Memory.

Blocul de calcul în rețelele RNN are o structură foarte simplă, cum ar fi, de exemplu, aplicarea asupra datelor de intrare a unei sigure funcții de tip tangentă hiperbolică. În cazul LSTM, pe de altă parte, blocul de calcul are o structură mai complexă. În locul

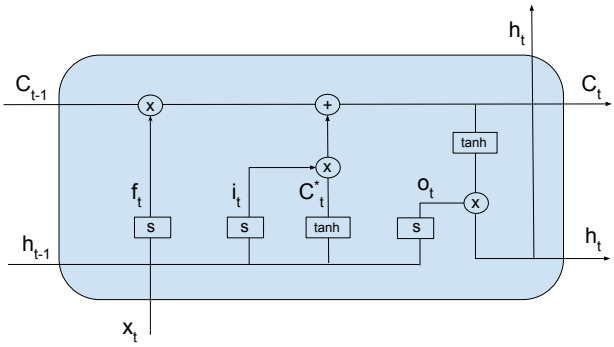

Figura I.2.1: O celulă de tip LSTM

unui singur strat ascuns, în cazul LSTM, sunt patru straturi care interacționează între ele. Figura I.2.1 descrie o astfel de celulă. C_{t-1} reprezintă starea celulei. Blocurile de tip s sunt funcții de tip sigmoid ($1/(1+e^{-x})$), $tanh$ este tangentă hiperbolică iar blocurile de tip $\times$ și $+$ reprezintă produsul scalar sau adunarea vectorilor de intrare.

Intrările h_{t-1} (starea latentă a sistemului) și x_t (datele de intrare propriu-zise) sunt trecute printr-o funcție de tip sigmoid al cărei rezultat este o valoare între zero și unu pentru fiecare valoare din starea C_{t-1}. Această operație poartă numele de *poartă de uitare* și decide ce informații din starea celulei sunt păstrate și cu ce pondere. Următorul pas decide ce informație este stocată în celulă. Inițial un strat de tip sigmoid denumit *poartă de intrare* decide ce valori vor fi actualizate. Ulterior, un strat de tip tangentă hiperbolică creează un vector de valori candidat C_t^*, care poate fi adăugat stării celulei. Ultima transformare combină rezultatul acestor două straturi. Ieșirea celulei, h_t (noua stare latentă a sistemului) se bazează pe o versiune filtrată a stării celulei. Primul strat în calculul ieșirii este reprezentat de o funcție sigmoid - *poarta de ieșire* - care decide ce parte din starea celulei va sta la baza ieșirii. Apoi se folosește un strat de tip tangentă hiperbolică pentru a scala valorile în intervalul $(-1, 1)$.

Există și alte variante pentru un bloc de tip LSTM, printre cele mai cunoscute fiind Gated Recurrent Unit (GRU) propusă de Cho *et al.* [19] și de Gert *et al.* [31].

Capitolul I.3

Generarea în paralel a numerelor aleatoare

Acest capitol descrie patru implementări posibile pentru generarea în paralel a numerelor pseudo-aleatoare, împreună cu avantajele și dezavantajele fiecăreia. Tehnicile descrise în continuare sunt folosite în capitolul III.2 pentru implementarea sistemului de decizie propus.

Calitatea unui generator de numere pseudo-aleatoare poate fi descrisă prin următoarele proprietăți:

- Instanțele generate sunt uniform distribuite. Pentru distribuții altele decât cea uniformă există diverse tehnici de generare bazate pe distribuția uniformă. [11].
- Instanțele generate nu sunt corelate.
- Generatorul nu are cicluri.
- Secvențele generate sunt reproductibile pe orice computer.
- Generatorul este rapid și consumă puțină memorie.

În cazul generatoarelor secvențiale aceste proprietăți sunt destul de evidente și, chiar dacă nu există generatoare care să le satisfacă pe toate în același timp, se poate obține un bun compromis [69]. Situația devine însă complicată în cazul generatoarelor paralele unde, pe lângă constrângerile menționate anterior mai trebuie satisfăcute și următoarele restricții:

[R1] Secvențele de numere generate nu trebuie să fie corelate.

[R2] Generatorul trebuie să scaleze ușor.

[R3] Fiecare worker trebuie să poată genera o nouă secvență aleatoare în mod independent de ceilalți workeri și de master.

Manager-Worker

Manager-Worker este o abordare centralizată a generării numerelor aleatoare. Din punct de vedere al implementării este de departe cea mai simplă strategie dar

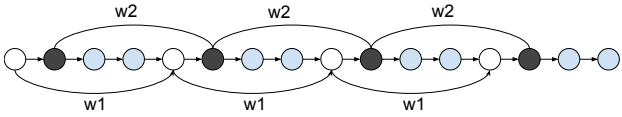

Figura I.3.1: Leapfrog - exemplu pentru patru Workeri. Sunt evidențiate valorile atribuite Workerului 1 și Workerului 2

devine din ce în ce mai puțin populară din cauza inconvenientelor majore de care dă dovadă [69]. Un "Manager" este însărcinat cu generarea numerelor aleatoare și cu distribuirea lor către "Workeri". Deși simplu de implementat nu respectă constrângerile [R1] și [R3] din lista prezentată anterior. Secvențele de numere pseudoaleatoare distribuite fiecărui Worker fac parte din aceeași secvență globală deci sunt corelate între ele. În lipsa unui Manager niciun Worker nu are acces la propria secvență de valori deci nu poate funcționa independent. Mai mult decât atât, dacă generarea este bazată pe cererile primite de la Workeri atunci procesul nu este unul reproductibil deoarece ordinea în care cererile ajung la Master nu este garantată ceea ce înseamnă că la execuții succesive aceeași valoare poate fi distribuită altui Worker.

Leapfrog

Metoda Leapfrog face parte din categoria generatoarelor paralele distribuite. Metoda este indicată când numărul de Workeri este cunoscut înainte de începerea execuției. Workerul de pe o poziție oarecare i "sare" $i-1$ poziții din secvența de numere aleatoare disponibilă. Pentru a genera această secvență, fiecare worker implementează o copie a generatorului. Există și variantă asemănătoare strategiei de tip Manager-Worker în care generarea propriu-zisă cade în sarcina unui Manager dar în acest caz, constrângerea [R3] din lista de mai sus nu mai este satisfăcută. Din cauza faptului că fiecare worker sare peste $i-1$ poziții în secvența aleatoare, valorile aflate la o distanță l în secvența centrală vor fi la o distanță l/W în secvența atribuită fiecărui Worker, unde W este numărul total de workeri. Apare astfel riscul de a crește gradul de corelație între valorile secvențelor atribuite Workerilor [80]. Un alt dezavantaj major al acestei metode este dat și de faptul că nu poate fi aplicată dacă numărul de Workeri este dinamic. Figura I.3.1 prezintă un exemplu de atribuire a valorilor pentru patru Workeri.

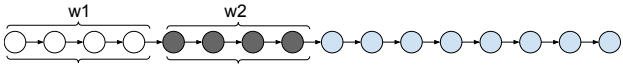

Figura I.3.2: Split Sequence - exemplu pentru patru Workeri. Sunt evidențiate valorile atribuite Workerului 1 și Workerului 2

Split Sequence

Acestă strategie constă în împărțirea deterministă a secvenței principale de numere aleatoare și atribuirea unei sub-secvențe fiecărui Worker. Spre deosebire de Leapfrog, numărul de Workeri poate fi dinamic dar numărul de valori necesare fiecărui Worker nu poate crește mai mult de un prag stabilit anterior - acest prag este dat de numărul de valori atribuite inițial Workerului respectiv. Dezavantajul principal al acestei abordări este că fiecare Worker trebuie să gliseze prin secvența de numere pseudo-aleatoare până când ajunge la poziția lui de start - acest proces poate fi consumator de timp așadar restricția [R2] menționată anterior este doar parțial satisfăcută.

Figura I.3.2 prezintă modul de implementare al unui generator de tip Split Sequence pentru cazul în care fiecare Worker necesită maxim patru valori aleatoare. Este evidențiată atribuirea valorilor pentru doi Workeri. În cazul prezentat, Workerul $w2$ trebuie să gliseze prin secvență patru poziții. Presupunând că fiecare Worker necesită patru valori atunci Workerul de pe o poziție oarecare i trebuie să gliseze $4(i-1)$ poziții în listă. Cu cât numărul de Workeri sau numărul de valori necesare fiecărui Worker crește, cu atât această glisare devine mai dăunătoare pentru performanța sistemului.

Parametrization

Tehnica Parametrizării constă în crearea și atribuirea unei instanțe ale aceluiași generator pentru fiecare Worker. Cazul ideal este acela în care fiecare generator folosește o valoare inițială diferită, ceea ce adaugă un nivel de independență crescut între Workeri. În cazul în care toate instanțele generatorului ar folosi aceeași valoare inițială, atunci constrângerea [R1] ar fi încălcată.

Capitolul I.4
Tehnici de optimizare a hiperparametrilor

Capitolul curent prezintă pe scurt tehnicile de optimizare a hiperparametrilor care sunt folosite în partea a III-a și anume Grid Search, Random Search, Nelder-Mead, Particle Swarm și Secvențele Sobol.

Grid Search (GS)

Algoritmul de tip Grid Search este un algoritm de căutare exhaustiv care constă în "reprezentarea" unei grile peste spațiul de căutare și evaluarea funcției obiectiv în fiecare nod al grilei. Nodul care generează cel mai bun rezultat este în final ales. Tehnica este similară cu alegerea manuală a valorilor inițiale pentru hiperparametri cu excepția faptului că induce un oarecare nivel de rigoare. Cel mai important parametru pentru Grid Search este numărul maxim de evaluări N. Pentru un spațiu d dimensional, se decide numărul de valori ce urmează a fi testate pentru fiecare dimensiune și se evaluează funcția obiectiv pentru fiecare combinație a acestor valori. Acest mod de împărțire a spațiului este extrem de ineficientă în cazul în care numărul de dimensiuni d este mare, caz în care, pentru a menține același număr maxim de iterații, N, numărul de puncte selectate pentru fiecare din dimensiuni este mic și astfel algoritmul tinde să ignore valorile cu potențial ridicat.

Random Search (RS)

Random Search (RS) este similar cu Grid Search cu excepția faptului că în loc să selecteze combinații de valori poziționate pe o grilă predefinită în spațiul de căutare, alege aleator aceste valori. În cadrul aceluiași buget computațional (număr de încercări N) RS tinde să obțină soluții mai bune comparativ cu GS sau căutarea manuală [10]. Pe lângă numărul maxim de încercări N, performanța algoritmului mai este influențată și de distribuțiile de probabilitate folosite pentru a genera valorile hiperparametrilor cât și de calitatea generatorului.

Algoritmul nu rulează neapărat până la epuizarea tuturor încercărilor N, existând posibilitatea să fie întrerupt în cazul în care anumite criterii de convergență sunt

satisfăcute.

Particle Swarm Optimization (PSO)

PSO [44] optimizează o funcție obiectiv prin încercări alternative de a îmbunătăți o soluție candidat în raport cu o anumită măsură a calității acestei soluții. Algoritmul pornește cu un set inițial de soluții candidat (particule) care se deplasează prin spațiul de căutare în încercarea de a optimiza funcția obiectiv. Aceste particule sunt asimilate unui roi (swarm). Fiecare particulă este caracterizată prin poziție și un vector viteză. Cu fiecare iterație, particulele se deplasează prin spațiul de căutare în conformitate cu o funcție care ia în considerare atât cea mai bună poziție a particulei cât și cea mai bună poziție globală, la nivel de roi. Atât poziția particulei cât și vectorul ei de viteză sunt actualizate.

Și în cazul acestui algoritm, numărul total de iterații N este limitat. Din cauza acestei limite, cu cât numărul de particule este mai mare, cu atât numărul de iterații pentru fiecare particulă este mai mic.

Nelder Mead (NM)

Nelder Mead este o metodă numerică aplicată destul de des [59] pentru a găsi punctul de extrem al unei funcții într-un spațiu multidimensional. Metoda este eficientă în special pentru funcții unimodale, fără variații bruște. Chiar dacă, de cele mai multe ori, funcția care caracterizează o problemă de optimizare a hiperparametrilor nu respectă aceste condiții și există astfel un risc considerabil ca algoritmul să se blocheze într-un optim local, acesta este totuși util datorită ratei de convergență foarte bune comparativ cu celelalte tehnici.

Pentru un spațiu d dimensional, NM menține un set de $d + 1$ puncte aranjate sub forma unui simplex și calculează valoarea funcției în fiecare din aceste puncte. Folosind valorile calculate, algoritmul extrapolează comportamentul funcției obiectiv în încercarea de a identifica un nou punct de test cu o valoare potențial mai bună. Algoritmul se oprește fie după un număr maxim de iterații, fie când o anumită condiție de convergență este satisfăcută. De cele mai multe ori condiția este ca simplexul să fie ”suficient de mic”.

Secvențe Sobol (SS)

Secvențele Sobol sunt secvențe de valori generate de algoritmul Sobol [78]. Proprietatea numerelor Sobol este aceea că oferă o discrepanță scăzută - acoperă spațiul de valori posibile într-un mod mai uniform comparativ cu variabilele aleatoare standard. Acest tip de acoperire rezultă într-o convergență superioară.

Pentru un hipercub unitate d-dimensional, I^d, și o funcție reală f integrabilă pe I^d o secvență Sobol este o secvență de puncte d-dimensionale $p_i, i = 1, \ldots, n$ astfel încât relația (I.4.1) să fie satisfăcută și convergența să fie cât mai rapidă.

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(p_i) = \int_0^1 \ldots \int_0^1 f(x_1, \ldots, x_n) dx_1, \ldots, dx_n \tag{I.4.1}$$

Pentru ca suma din relația (I.4.1) să conveargă la integrala din membrul drept punctele p_i ar trebui să acopere I^d astfel încât să minimizeze golurile dintre ele. O altă proprietate necesară este ca proiecția punctelor pe fiecare din fețele hipercubului să minimizeze de asemenea golurile [78]. Acest al doilea criteriu este motivul pentru care punctele distribuite omogen în interiorul hipercubului nu reprezintă o secvență Sobol (în acest caz, al punctelor distribuite omogen, proiecția mai multor puncte pe fețele hipercubului s-ar suprapune).

Convergența superioară a secvențelor Sobol comparativ cu generarea de variabile aleatoare dintr-o distribuție uniformă, le face un bun candidat pentru utilizarea în algoritmii de optimizare a hiperparametrilor, o mai bună acoperire a spațiului de valori posibile ducând la o probabilitate mai mare de a identifica o valoare optimă pentru funcția obiectiv.

Capitolul I.5

Apache Spark

Acest capitol prezintă noțiuni de bază legate de Apache Spark. Apache Spark, introdus în 2010 de Zaharia *et al.* [87], stă la baza implementării sistemelor de recomandare descrise în partea a II-a.

Apache Spark este un ansamblu de biblioteci software dedicate procesării distribuite de date. Principalele sale caracteristici sunt viteza, ușurința în utilizare și flexibilitatea [86]. Datorită acestor calități Spark este extrem de popular iar rata lui de adopție este în continuă creștere[1]. Un alt aspect important de menționat este că acești operatori sunt disponibili în mai multe limbaje de programare: Scala, Java, Python și R.

Bibliotecile software dedicate procesării de date sunt doar o parte din ecosistemul Spark. Acesta înglobează, de asemenea, un sistem de stocare de date distribuit (HDFS), un sistem de gestiune a clusterelor de servere precum și posibilitatea de a lucra cu date în formate variate, atât binare cât și columnare.

În 2015 IBM anunță o investiție masivă[2] în Spark, pe care în cataloghează ca fiind probabil "cel mai important proiect de tip software open source destinat procesării datelor din ultima decadă".

Următoarele secțiuni prezintă pe scurt conceptele care stau la baza arhitecturii Spark.

Arhitectura unei aplicații Spark

Spark este, în esență, un sistem distribuit care a fost conceput pentru a procesa cât mai eficient și mai rapid un volum mare de date. Acest sistem distribuit este instalat, în general, pe mai multe servere. Ansamblul acestor servere constituie un cluster Spark. Dimensiunea unui astfel de cluster poate să varieze de la câteva servere până la câteva mii (cel mai mare cluster Spark cunoscut numără mai mult de 8000 de servere [63]).

[1] https://www.datanami.com/2019/03/08/a-decade-later-apache-spark-still-going-strong/
[2] https://www-03.ibm.com/press/us/en/pressrelease/47107.wss

O aplicație tipică Spark este alcătuită din două componente. Prima componentă constă în logica de procesare a datelor folosind operatorii puși la dispoziție de Spark. Cealaltă componentă este driverul Spark. Acest driver este coordonatorul aplicației Spark și are ca rol interacțiunea cu managerul clusterului pentru a identifica pe care din servere poate rula codul de procesare a datelor. Pe fiecare din workerii dintr-un cluster, managerul clusterului lansează, la cererea driverului, un proces de tip executor. Driverul distribuie apoi sarcinile fiecărui executor și eventual colectează rezultatele execuției. Fiecare executor Spark este un proces JVM (Spark este implementat în Scala, care este un limbaj bazat pe JVM). Acest proces este alocat exclusiv aplicației Spark care l-a generat, astfel o aplicație mai puțin performantă nu va influența negativ celelalte aplicații care rulează pe același cluster. Durata de viață a unui executor Spark este aceeași cu durata de viață a aplicației care l-a creat. O aplicație Spark constă într-un singur driver (driverul joacă rolul masterului din perspectiva unei arhitecturi master-slave) și unul sau mai mulți executori. Fiecare executor execută procesările de date care îi sunt alocate sub forma de sarcini. Fiecare sarcină este executată pe un nucleu separat, Spark putând astfel paraleliza execuția acestora în funcție de numărul de nuclee disponibile în cluster.

Ecosistemul Spark

Spark furnizează un adevărat ecosistem în domeniul procesării distribuite de date. Fundația acestui ecosistem constă în modulul de bază: Spark Core, care oferă funcționalitatea necesară pentru a gestiona și executa aplicații distribuite precum și o abstractizare a modului în care sunt reprezentate datele manipulate de astfel de aplicații. Datele sunt modelate sub forma unor structuri numite ”resilient distributed datasets” - RDD - structuri de date ce permit execuția de operații specifice colecțiilor standard dar într-o manieră distribuită. Pe baza nucleului Spark sunt oferite mai multe componente, fiecare cu o zonă de aplicabilitate specifică. Dintre acestea, cel mai important în contextul prezentei teze este Spark MLlib: o bibliotecă Spark dedicată învățării automate. Oferă mai mult de 50 de algoritmi precum și funcționalități transversale ca persistarea modelelor, curățarea datelor, evaluarea rezultatelor etc.

Spark în contextul învățării automate

Construcția unui model de tip ML constă, de obicei, într-un set de pași care rulează secvențial. Anumiți pași trebuie repetați cu scopul de a îmbunătăți modelul până când un anumit criteriu de calitate este satisfăcut sau îmbunătățirea nu mai este posibilă. Spark, prin biblioteca MLlib, oferă un set de abstracții

care simplifică definirea și execuția acestor pași. Astfel, sunt oferite abstracții pentru curățarea datelor, identificarea caracteristicilor, antrenarea și optimizarea modelului, evaluarea performanței precum și posibilitatea de a organiza aceste abstracții într-o secvență bine definită de pași. Modelul MLlib este inspirat din, probabil cea mai utilizată bibliotecă ML, scikit-learn [68]. Astfel, în MLlib sunt patru categorii de abstracții.

- Transformatoare - destinate transformării datelor dintr-un format în altul. Câteva exemple de astfel de transformatoare sunt SQLTransformer, MinMaxScaler, Normalizer, OneHotEncoder, Tokenizer etc. (numele descriu destul de bine funcționalitatea oferită de fiecare).
- Estimatoare - abstracții ale algoritmilor de ML, de exemplu: LogisticRegression, DecisionTreeClassifier, LinearRegression, KMeans etc. Un estimator construiește un model plecând de la datele de intrare.
- Evaluatoare - sunt destinate evaluării performației modelelor create de estimatoare.
- Fluxuri - abstracții care permit definirea de secvențe de pași, unde fiecare pas este fie un transformator fie un evaluator. Un flux este totodată și un estimator, rezultatul lui fiind, de asemenea, un model.

Folosind MLlib pot fi implementați algoritmi standard din învățarea automată capabili însă (având suportul Spark Core) de a rula pe clustere de dimensiuni impresionante și astfel capabili de a gestiona cantități impresionante de date.

Partea II

Atribuirea automată a erorilor în proiectele mari de tip software open source

Această secțiune tratează problema atribuirii automate a erorilor în proiectele mari de tip software open source și constă din trei capitole. Primul capitol este unul introductiv unde este prezentată problema și stadiul actual al cercetării. Capitolul II.2 prezintă o primă propunere pentru un sistem de recomandare în domeniul atribuirii automate a erorilor. Este vorba de o implementare paralelă pe o arhitectură de tip cloud (prima de acest gen, din cunoștințele mele) al cărei nucleu este constituit de un model de tip SVM (Support Vector Machines - Mașini cu Suport Vectorial). Sistemul de recomandare propus obține rezultate similare celor raportate în literatura de specialitate în contextul unei scalabilități net superioare sistemelor desktop existente. Capitolul II.3 prezintă o primă încercare de a folosi tehnici de învățare profundă în zona atribuirii automate a erorilor. Implementarea propusă este de asemenea una paralelă, scalabilă, bazată pe o arhitectură de tip cloud. Am folosit pentru acest sistem de recomandare, alternativ, CNN (Convolutional Neural Networks - Rețele neurale convoluționale) și LSTM (Long Short-Term Memory - Rețele neurale recurente cu memorie pe termen lung). Rezultatele obținute cu ajutorul CNN sunt la egalitate cu cele mai bune din domeniu și sunt obținute și de această dată în contextul unei scalabilități mult superioare implementărilor anterioare.

Capitolul II.1
Stadiul actual al cercetării

Capitolul curent descrie pe scurt problema atribuirii erorilor cu accent pe importanța acesteia în proiectele de tip software open source. Este menționat, de asemenea, și modul în care o astfel de problemă poate fi tratată din perspectiva învățării automate. Sunt prezentate câteva din lucrările semnificative în domeniul asignării automate de erori cu accent pe aplicarea tehnicilor de învățare automată în rezolvarea acestui tip de problemă. Totodată sunt prezentate și încercările de a integra algoritmi din zona specifică a învățării profunde în dezvoltarea unor sisteme inteligente de decizie destinate acestui tip de asignare automată.

Marea majoritate a proiectelor software folosesc un sistem de urmărire a problemelor (ITS - Issue Tracking System) pentru a organiza procesul de rezolvare al acestora. Raportele introduse într-un ITS urmează un flux de lucru standard. Odată introdus, un raport este ”triat” de către un manager de proiect. Procesul de triere implică mai multe etape incluzând determinarea dacă raportul este unul valid sau un duplicat al unui raport deja existent, dacă a fost corect clasificat per produs, componentă sau subcomponentă a produsului cât și atribuirea persoanei cea mai potrivită pentru a trata raportul respectiv. Această triere a rapoartelor este o activitate obligatorie în utilizarea oricărui ITS [72] și, în cazul proiectelor de dimensiuni considerabile, poate fi mare consumatoare de timp din cauza numărului mare de rapoarte noi introduse per unitate de timp [6].

Majoritatea efortului de cercetare în domeniul trierii automate a erorilor se concentrează pe decizia de atribuire a celei mai potrivite persoane pentru a trata problema raportată, cu scopul de a sugera varianta (variantele) optime de asignare. Acest proces este unul complex deoarece pe lângă dificultatea aferentă identificării persoanei potrivite pentru a repara o anumită eroare trebuie luate în considerare și aspecte legate de încărcarea și disponibilitatea acesteia. Problema devine cu atât mai importantă în contextul proiectelor de tip software open source de mari dimensiuni deoarece nu există o triere prealabilă a dezvoltatorilor - oricine dorește poate contribui la un astfel de proiect, cu condiția să dezvolte cod în conformitate cu standardele de calitate ale proiectului respectiv. Ușurința cu care cineva poate deveni parte a unei echipe de dezvoltare pentru un proiect de acest gen coroborat

cu interesul ridicat pentru domeniul software-ului open source duce la un număr foarte mare de dezvoltatori.

Din perspectiva învățării automate (ML), problema atribuirii erorilor poate fi privită ca o problemă de clasificare supervizată. Mai specific, datele de intrare sunt constituite de informația textuală și categorială corespunzătore unui raport iar clasele de ieșire sunt identificatorii dezvoltatorilor. O serie de clasificatorare au fost folosite în acest context, incluzând Naïve Bayes, Bayesian Networks, C4.5, Support Vector Machines (SVM), și k-Nearest Neighbors (kNN). Pe de altă parte, tehnicile de învățare profundă (DL) deși oferă rezultate promițătoare intr-o arie vastă de aplicabilitate [32] au o rată de adopție scăzută în domeniul sistemelor de decizie pentru asignarea automată de erori. Acest lucru este cauzat cel mai probabil de timpul lung necesar pentru antrenarea acestor modele și de complexitatea ridicată a arhitecturilor cu consecințe negative directe în scalabilitatea lor.

În contextul în care bazele de date cu rapoarte de erori continuă să crească (mai mult de 14.000 de intrări noi și aproape 2.000 de profile de dezvoltatori au fost adăugate pentru Mozilla în primele două luni ale anului 2016, generând un total de peste 500.000 de intrări și mai mult de 50.000 de profile create în ultimii ani[1]), capacitatea de a scala sistemul pentru o cantitate de date în continuă creștere devine critică, în special în etapa de antrenare.

Diverse abordări pentru crearea sistemelor de recomandare pentru asignarea automată a erorilor au fost investigate, majoritatea concentrându-se pe analiza informației textuale și categoriale folosind fie tehnici specifice de extragere și recunoaștere a informației fie învățarea automată [15]. Secțiunea curentă menționează exclusiv lucrările din domeniu care folosesc tehnici de învățare automată.

Primele încercări de folosire a învățării automate în domeniul asignării de erori au fost făcute de Cubranic și Murphy [22]. Autorii au raportat o acuratețe a clasificării de 30% pe 15,859 rapoarte de eroare aferente proiectului Eclipse. Prin tehnici de filtrare a datelor neesențiale Anvik [6], [7] a reușit îmbunătățirea acurateței clasificatorului la mai mult de 50%. Spre deosebire de incercările originare care s-au rezumat la o tehnică specifică, rezultatele mai multor modele de de algoritmi de învățare automată(Naïve Bayes, SVM, și C4.5) au fost comparate de Anvik *et al.* [6], SVM obținând cea mai ridicată performanță.

Shokripour *et al.* [75] au aplicat o filtrare mai agresivă a datelor de intrare în funcție de părțile de vorbire. Abordarea lor constă în extragerea exclusivă a substantivelor pentru a putea identifica și prezice fișierele sursă care vor fi potențial modificate

[1] Date extrase dintr-un export al bazei de date Mozilla din 4 Martie 2016

de rezolvarea fiecărei erori raportate și, în funcție de ele, pentru a decide ce dezvoltator să recomande, bazat pe intervențiile precedente asupra acestor fișiere sursă. Similar cu Shokripour *et al.*, Florea *et al.* [29] folosește exclusiv substantivele dar în combinație cu tehnici de reducere a dimensionalității precum Chi^2 sau Latent Dirichlet Allocation (LDA).

Deși diverse tehnici de învățare automată au fost aplicate în zona asignării automate a erorilor: Naïve Bayes [58], SVM [29], și C4.5 [6] atât independent cât și în combinație cu algoritmi de reducere a dimensionalității cum ar fi: LSI [1], Chi^2 [8,29] și LDA [29, 61], cu excepția abordării din [28], Învățarea profundă nu a devenit încă populară în acest domeniu. Acest lucru poate fi cauzat de complexitatea sistemelor de acest tip, de numărul mare de parametri necesari pentru a construi astfel de modele și de timpul, în general mare, necesar antrenării.

Cu toate că domeniul specific al asignării automate de erori nu profită încă pe deplin de avantajele învățării profunde, aceasta devine, progresiv, din ce în ce mai populară în zona sistemelor inteligente de decizie, în general. Considerând nivelul ridicat de complexitate a modelelor de tip învățare profundă dar și cantitatea de date în continuă creștere disponibilă ca intrare pentru un sistem inteligent de decizie în domeniul atribuirii automate a erorilor, devine aproape obligatorie folosirea resurselor unui cluster ca suport pentru o implementare distribuită a algoritmului de antrenare pentru un astfel de sistem. Am propus o primă implementare de acest gen în Florea *et al.* [29] unde am folosit o versiune distribuită de SVM pe un cluster Apache Spark. În contextul în care Apache Spark oferă un API dens și intuitiv pentru implementarea de aplicații paralele, combinarea Spark cu învățarea profundă este un trend din ce în ce mai des întâlnit în cercetarea actuală. Stoica *et al.* [57] au dezvoltat SparkNet, o bibliotecă software cadru care folosește Apache Spark pentru a antrena mai multe instanțe de modele Caffe, distribuite pe un cluster, folosind o tehnică de paralelizare a datelor[2]. Încercări similare de a folosi Apache Spark pentru a paraleliza algoritmi de tip învățare profundă au fost raportate și de Abu Alsheikh [4] al cărui efort se concentrează pe domeniul analizei datelor mobile. Strict în contextul recomandării asignării de erori am propus o primă abordare pentru aplicarea tehnicilor de învățare profundă, mai precis CNN si LSTM, într-un context distribuit bazat pe Apache Spark în Florea *et al.* [28].

[2] https://en.wikipedia.org/wiki/Data_parallelism

Capitolul II.2

Implementarea unui sistem de recomandare de tip cluster pentru asignarea automată a erorilor în proiectele mari de tip software open source folosind Apache Spark

Acest capitol are la bază rezultatele prezentate în cadrul ”International Conference on Artificial Intelligence and Soft Computing” în 2017 și publicate în Springer, ”Lecture Notes in Computer Science” volumul 10246 cu titlul: ”Spark-based cluster implementation of a bug report assignment recommender system” [29]. Capitolul prezintă o propunere de implementare a unui sistem de recomandare de tip cluster pentru asignarea automată a erorilor în proiectele mari de tip software open source folosind Apache Spark. Inițial este descris sistemul de recomandare, apoi sunt prezentate rezultatele obținute de acest sistem pe trei seturi de date corespunzătoare unor proiecte reale. Capitolul continuă cu o discuție asupra performanței și scalabilității implementării propuse.

II.2.1 Sistemul de recomandare

Scopul sistemului de recomandare propus este de a sugerea cel mai potrivit dezvoltator care poate rezolva o problemă raportată, plecând de la datele de intrare ale acestei probleme. Secțiunea curentă introduce sistemul de recomandare, descrie seturile de date de intrare, procesul de curățare a datelor, pașii aplicați pentru pre-procesarea datelor, algoritmul de antrenare și implementarea propriu-zisă.

II.2.1.1 Seturile de date

Sunt folosite următoarele seturi de date de intrare:

- Eclipse Bugzilla[1] set de date disponibil pe site-ul MSR 2011.[2]

[1] https://bugs.eclipse.org/bugs/
[2] http://2011.msrconf.org/msr-challenge.html

- Netbeans Bugzilla[3] set de date, de asemenea disponibil pe site-ul MSR 2011.
- Mozilla Bugzilla[4] pus la dispoziție, la cerere, de către Mozilla Foundation.

Toate cele trei proiecte folosesc Bugzilla[5] ca sistem de urmărire a problemelor. Pentru toate cele trei seturi de date este utilizat un export al bazei de date MySQL care conține datele din Bugzilla, formatul datelor este, în consecință, același.

Pentru implementarea sistemului de recomandare sunt interogate date din tabelele `bugs`, `duplicates`, `longdescs` și `bugs_activity`. Două tabele suplimentare, `fileddefs` și `attachments` sunt folosite exclusiv pentru pre-filtrare și nu intervin în datele de antrenare.

Odată introdus în sistem, un raport urmează variații ale a unui flux de lucru clasic, în funcție de regulile specifice ale proiectului. El trece prin mai multe etape și, în final, este închis. Închiderea unui raport poate fi făcută prin selectarea unei rezoluții specifice, cum ar fi:

- FIXED - dacă eroare a fost rezolvată
- DUPLICATE - în cazul intrărilor duplicate
- WORKSFORME, WONTFIX sau INVALID - dacă se decide că raportul în cauză nu reprezintă o problemă reală
- LATER - dacă este amânată rezolvarea

Pentru sistemul de recomandare am extras informații categoriale din tabela `bugs`, incluzând data la care un raport a fost creat, statusul acestuia, identificatorul produsului și cel al categoriei. Informația textuală cu privire la eroare este extrasă din tabela `longdescs`. Interoghez tabela `bugs_activity` pentru a determina ce dezvoltator a marcat raportul ca FIXED. Folosesc de asemenea informația din tabela `duplicates` pentru a exclude intrările duplicate din etapa de antrenare deoarece nu contribuie la îmbunătățirea predicției.

II.2.1.2 Curățarea datelor

Pentru a fi util, un sistem de recomandare pentru asignarea automată a erorilor trebuie să sugereze exclusiv dezvoltatori care sunt activi în proiectul respectiv. De aceea, filtrez datele originale de intrare și selectez exclusiv dezvoltatorii cu o medie de cel puțin trei erori reparate pe lună, în ultimele trei luni. Această filtrare

[3] https://netbeans.org/bugzilla/
[4] https://bugzilla.mozilla.org/
[5] https://www.bugzilla.org/docs/2.18/html/dbdoc.html

contribuie la eliminarea persoanelor care intervin exclusiv ocazional în proiect. Din datele deja pre-filtrate, în forma `(developerId, fixedBugs)`, calculez media și abaterea standard pentru `fixedBugs` și din nou elimin dezvoltatorii care au marcat ca FIXED mai mult de `mean+2*stddev` erori. Întregul proces are ca scop excluderea membrilor proiectului care par a avea rol de manager de proiect sau care nu au avut activitate recentă.

Pentru fiecare raport marcat ca FIXED, dezvoltatorul care l-a corectat efectiv este sau cel care apare în câmpul **`assigned_to`** din tabela `bugs` sau acela care a marcat efectiv raportul ca FIXED confirm cu tabela `bugs_activity`. Anvik *et al.* [7] consideră că, din punct de vedere euristic, a doua abordare este mai potrivită. Din acest motiv prezentul sistem de recomandare folosește această abordare.

Pe lângă informația textuală din titlul, descrierea și comentariile fiecărui raport, sunt folosite, de asemenea, și informațiile referitoare la componenta și produsul fiecărui raport, prezente în câmpurile **`component_id`** și **`product_id`** din tabela `bugs`. O caracteristică unică a abordării prezentate este aplicarea unor coeficienți de pondere acestor vectori binari. Valorile optime pentru acești coeficienți sunt determinate prin intermediul Grid Search [10], folosind 100 de execuții cu un spațiu al valorilor între unu și 100 pentru fiecare parametru, urmată de o a doua rundă de teste în intervalul centrat în jurul rezultatelor obținute în urma rundei inițiale de optimizare.

II.2.1.3 Preprocesare și reducerea dimensionalității

Similar cu abordarea din Shokripour *et al.* [75], sistemul de recomandare propus folosește algoritmul de identificare a părților de vorbire Stanford [83] pentru a filtra informația textuală și pentru a extrage și păstra exclusiv substantivele. Faza de preprocesare constă în convertirea datelor textuale în date numerice folosind TF/IDF [70]. Am exclus acei termeni care apar în mai puțin de două sau în mai mult de 15% din numărul total de documente. Pentru antrenare, validare și testare am folosit datele din cele mai recente 240 de zile deoarece informațiile cele mai recente sunt, implicit, și cele mai relevante. Sistemul de recomandare folosește, alternativ, fie întregul spațiu de caracteristici obținut, fie una din următoarele două tehnici de reducere a dimensionalității: Chi^2 [85] sau Alocarea Latentă Dirichlet (LDA) [12].

II.2.1.4 Antrenarea sistemului de recomandare

În contextul menționat de Ahsan *et al.* [1, 7] care au demonstrat ca SVM-ul obține performanțele cele mai bune între tehnicile clasice de învățare automată am decis

folosirea unui clasificator de tip SVM ca nucleu pentru sistemul de recomandare propus. Am păstrat cele mai recente 10% intrări ca date de test iar restul de 90% pentru antrenare și validare (80% antrenare și 10% validare).

Am antrenat câte un model SVM liniar [21] pentru fiecare clasă (dezvoltator), aplicând o strategie de tip one-versus-all. Rezultatul predicției este o singură clasă (dezvoltator) și anume cea a cărei probabilitate este cea mare. Pentru evaluarea sistemului este calculată precizia, senzitivitatea și F1-measure per clasă [79] și utilizată media lor ponderată cu frecvența fiecărei clase. Antrenarea modelului este făcută cu un algoritm de tip Stochastic Gradient Descent cu regularizare de tip L2, pasul (rata de învățare) de unu, cel mult 100 de iterații, valoarea parametrului de regularizare de 0,01 și o toleranță de 0,001[6]. Valorile pentru acești parametri au fost identificate folosind Grid Search.

II.2.1.5 Detalii de implementare

Implementarea este făcută cu ajutorul platformei de tip cloud Apache Spark[7]. Sistemul de recomandare foloseste implementarea SVM de tip LIBLINEAR [25] din MLLIB[8]. Codul este implementat folosind Scala 1.10.6[9] - cea mai recentă versiune la momentul implementării. Codul este disponibil public pe GitHub[10]. Testele au fost efectuate pe un cluster de tip Google DataProc[11], constând într-un nod de tip master și trei noduri de tip worker, toate cele patru mașini de tip n1-highmem-2 (2 vCPU, 13.0 GB memorie RAM) cu 100 GB spațiu pe disc.

II.2.2 Rezultate și discuții

Secțiunea curentă discută rezultatele obținute de sistemul de recomandare proiectat, cu accent asupra influenței aduse de cele două tehnici de reducere a dimensionalității. Secțiunea se încheie cu comparația rezultatelor obținute în raport cu abordările anterioare asupra problemei atât din punct de vedere al performanței recomandării cât și în ceea ce privește scalabilitatea sistemului.

[6] https://github.com/apache/spark/blob/branch-1.6/mllib/src/main/scala/org/apache/spark/mllib/optimization/GradientDescent.scala#L81

[7] http://spark.apache.org/

[8] http://spark.apache.org/mllib/

[9] http://www.scala-lang.org/

[10] https://github.com/acflorea/columbugus

[11] https://cloud.google.com/dataproc/

Pentru seturile de date de test, sistemul de recomandare a obținut cele mai bune rezultate în următoarele configurații:

- Eclipse - Precizia optimă 0,79, senzitivitatea optimă 0,77 și F1 optim 0,76
- Netbeans - Precizia optimă 0,89, senzitivitatea optimă 0,88 și F1 optim 0,88
- Mozilla - Precizia optimă 0,77, senzitivitatea optimă 0,75 și F1 optim 0,73

Reducerea dimensionalității induce o penalitate relativ importantă în performanța sistemului de recomandare dar prin reducerea spațiului de intrare permite antrenarea clasificatorului într-un timp rezonabil pe o singură stație de lucru, fără a mai fi necesar un cluster. Pentru Chi^2, orice număr mai mic de 1.000 de dimensiuni are influență negativă asupra preciziei clasificatorului. Cu 1.000 de dimensiuni sistemul rămâne stabil, obținând o scădere a preciziei cu mai puțin de 0,01 pentru Netbeans și Mozilla și chiar o îmbunătățire în cazul Eclipse, relativ la modelul fără reducere a dimensionalității. Pentru LDA, pe de altă parte, precizia scade indiferent de numărul de dimensiuni (aceste dimensiuni poată numele de topicuri în cazul LDA) ales. Numărul optim de topicuri este de asemenea în jurul lui 1.000.

II.2.2.1 Comparație cu abordările anterioare. Aspecte legate de scalabilitate

Am comparat rezultatele obținute cu implementări anterioare de sisteme de recomandare pentru asignarea automată a erorilor care au la bază SVM. Tabela II.2.1 arată rezultatele acestei comparații, cu cele mai bune rezultate marcate îngroșat. Implementarea propusă de mine obține rezultate similare cu implementările anterioare în ceea ce privește precizia dar valori net superioare pentru senzitivitate, pentru seturile de date Eclipse și Mozilla. Nu am reușit să identific în literatura de specialitate sisteme de recomandare bazate pe SVM care să folosească setul de date Netbeans.

Timpul total de creare (antrenare plus testare) pentru un sistem de recomandare folosind abordarea descrisă anterior este de mai puțin de 10 minute folosind arhitectura propusă în Secțiunea II.2.1.5, fără niciun compromis în ceea ce privește performanța sistemului. În contextul în care nu am identificat în literatură valori raportate pentru timpul necesar creării acestui gen de sistem, nu am avut o referința pentru a compara performanța obținută. În ciuda acestui fapt, consider că valorile obținute sunt foarte promițătoare.

Fiind vorba de un sistem implementat pe un cluster, am testat, de asemenea, scalabilitatea acestuia, folosind de la două la șase nuclee de calcul pe o cantitate de date în creștere. Tabela II.2.2 arată valorile speedup-ului

Tabela II.2.1: Precizia și senzitivitatea - Comparație cu implementări anterioare bazate pe SVM

	Eclipse Precizie/senzitivitate	Mozilla Precizie/senzitivitate
Implementarea propusă	0.78 / **0.77**	**0.76** / **0.75**
Nucleul Weka RBF	0.72 / 0.74	0.71 / 0.67
”One prediction” [6]	0.86 / 0.12	0.64 / 0.02
”Three predictions” [6]	0.77 / 0.32	0.53 / 0.06
”One prediction” [7]	**0.97** / 0.18	0.70 / 0.01
”Three predictions” [7]	0.79 / 0.41	0.64 / 0.03

(i.e., $\frac{\textit{timpul necesar unei execuții secvențiale}}{\textit{timpul necesar unei execuții paralele}}$) pentru faza de antrenare a sistemului de recomandare.

Tabela II.2.2: Speedup-ul pentru diferite dimensiuni ale setului de date de antrenare

Dimensiunea setului de date	2 nuclee	3 nuclee	4 nuclee	5 nuclee	6 nuclee
2,000,000	1.97	2.86	3.85	4.41	4.89
900,000	1.97	2.58	3.32	3.99	4.12
90,000	1.95	2.59	3.41	3.91	4.12
30,000	2.11	2.6	3.16	3.33	2.86

II.2.3 Concluzii

Din cele cunoscute de mine la momentul creării sistemului de recomandare, acesta este primul de acest gen implementat pe o platformă de tip cloud (Apache Spark găzduit pe o arhitectură de tip Google Cloud DataProc). Sistemul de recomandare propus oferă o alternativă rapidă și scalabilă pentru implementările existente de tip desktop și obține performanțe similare în ceea ce privește precizia și senzitivitatea pe trei seturi de date reale provenind de la unele din cele mai cunoscute proiecte de tip software open source.

Capitolul II.3

Implementarea paralelă a unui sistem de recomandare pentru asignarea automată a erorilor în proiectele mari de tip software open source folosind tehnici de învățare profundă

Rezultatele din acest capitol au la bază lucrarea prezentată în cadrul conferinței ”International Conference on Artificial Neural Networks” în 2017 și publicate în Springer, ”Lecture Notes in Computer Science” volumul 10614 cu titlul: ”Parallel Implementation of a Bug Report Assignment Recommender Using Deep Learning” [28]. Capitolul curent descrie un model de sistem de recomandare pentru asignarea automată a erorilor în proiecte mari de tip software open source. Sistemul de recomandare propus se bazează pe tehnici specifice de tip învățare profundă. Prima secțiune descrie sistemul de recomandare în ansamblu. În Secțiunea II.3.2 sunt prezentate rezultatele obținute pe trei seturile de date pentru proiectele Netbeans, Eclipse și Mozilla. Capitolul se încheie cu o secțiune de concluzii.

II.3.1 Sistemul de recomandare

Această secțiune prezintă detaliile sistemul de recomandare propus. Datele de intrare provin din exporturi MySQL ale bazelor de date cu rapoarte de eroare în format Bugzilla. Acestea sunt inițial filtrate și preprocesate conform detaliilor prezentate în Secțiunea II.3.1.1. Rezultatele obținute în urma preprocesării sunt folosite ca intrare pentru un model de tip Paragraph2Vec [49]. Modelul astfel rezultat determină intrarea pentru un clasificator de tip învățare profundă, mai exact, fie o rețea convoluțională (CNN) fie o rețea Long Short Term Memory (LSTM) [32].

Similar cu abordarea prezentată în capitolul precedent, pentru a evalua performanțele sistemului de recomandare am folosit date reale corespunzătoare proiectelor **Eclipse**, **Netbeans** și **Mozilla**.

II.3.1.1 Pregătirea datelor

Pentru curățarea datelor am aplicat un proces similar cu cel descris de Anvik și Murphy [7] și Florea *et al.* [29]. Din toate rapoartele cu statutul CLOSED (`bugs.bug_status = `CLOSED'`) am considerat exclusiv pe cele marcate ca FIXED (`bugs.resolution = `FIXED'`). Am considerat că dezvoltatorul care a corectat o eroare este cel care a marcat raportul aferent ca FIXED în tabela `bugs_activity`. Pentru a putea prezice eficient un dezvoltator care ar putea să trateze cu succes eroarea raportată am restrâns baza de predicție doar la dezvoltatorii activi în proiect - pentru ca un dezvoltator să fie considerat activ, acesta trebuie să fi rezolvat cel puțin trei rapoarte pe lună în ultimele trei luni. Am eliminat astfel dezvoltatorii cu o activitate punctuală pe proiect și deci cu șanse mici să mai intervină în viitor. Am exclus, de asemenea, persoanele cu o frecvență excepțional de ridicată a rapoartelor marcate ca FIXED (mai mult de $mean + 2 * stddev$, unde $mean$ și $stddev$ sunt media și abaterea standard pentru numărul de rapoarte tratate per utilizator) plecând de la premiza că acestea joacă mai degrabă un rol de Manager în proiect.

Pentru fiecare raport am folosit o versiune normalizată a reprezentării lui textuale, obținută prin convertirea textului la litere mici exclusiv, prin eliminarea oricărui caracter non-alfanumeric și prin concatenarea textului din titlu, descriere și comentarii. Pe lângă datele textuale am folosit, de asemenea, informația disponibilă în câmpurile `component_id`, `product_id` și `bug_severity` ca variabile categoriale transformate în vectori binari printr-o metodă de tip one-hot [36].

Am folosit versiunea de tip Distributed Bag of Words pentru Paragraph2Vec(PV-DBOW) [49] pentru a obține reprezentarea numerică a datelor textuale. Pentru antrenare, validare și testare, am folosit exclusiv rapoartele marcate ca FIXED în cele mai recente 240 de zile, considerând că datele mai recente sunt automat mai relevante.

II.3.1.2 Antrenarea sistemului de recomandare

Am ordonat seturile de date în funcție de data ultimei modificări făcute și le-am împărțit în 80% pentru antrenare, 10% pentru validare și 10% (cele mai recente) pentru testare. Am folosit un cluster Apache Spark[1] pentru a antrena atât o rețea neurală de tip CNN cât și una de tip LSTM.

Pentru a induce datelor un nivel mai ridicat de spațialitate între cuvintele dintr-un paragraf, în cazul CNN, am împărțit reprezentarea textuală a rapoartelor

[1] http://spark.apache.org/

în n secvențe egale și am calculat reprezentarea lor prin PV-DBOW individual. Arhitectura aleasă pentru rețeaua CNN constă în două perechi de straturi, unul convoluțional (($C(k_i, 1)(1, 1)f_i$) și celălalt de tip average pooling, urmate de două straturi dense. În cazul sistemului de recomandare propus, am folosit $n = 5$ și $k_1 = k_2 = 2$. Pentru primul strat dens am folosit 500 de neuroni și o funcție de activare de tip ReLU ($f(x) = max(0, x)$). Pentru cel de-al doilea strat dens am folosit o funcție de activare de tip softmax $\left(\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}\right)$.

Pentru LSTM nu este necesară o preprocesare suplimentară, ca în cazul CNN. Datele obținute în urma aplicării PV-DBOW sunt introduse într-o rețea recurentă cu două straturi. Primul strat este un strat "Graves" RNN [33] unidirecțional cu 250 de neuroni de ieșire, un dropout de 0.5 și funcția de activare de tip softsign $\left(f(x) = \frac{x}{1+|x|}\right)$. Stratul de ieșire folosește o funcție de activare softmax.

II.3.1.3 Detalii de implementare

Sistemul de recomandare este implementat în Scala 2.11.8 (cea mai recentă versiune la momentul implementării) cu excepția modulului de curățare a datelor care este implementat în Python 2.7.11. Am folosit Deeplearning4j[2] pentru a construi și antrena rețelele neurale. Am antrenat atât rețeaua de tip CNN cât și pe cea de tip LSTM folosind Apache Spark. Fiecare cluster Spark folosit constă într-un nod de tip master și 4, 8 sau 12 workeri. Am antrenat sistemul de recomandare pe două servere de tip IBM Power8 8001-22C (Briggs) cu sistem de operare Ubuntu 16.04 LTS, folosind versiunea I.B.M de Spark 1.6.3 (la momentul implementării, cea mai recentă versiune disponibilă). Codul este disponibil public pe GitHub[3].

II.3.2 Rezultate

Tabelele II.3.1, II.3.1 și II.3.3 arată valorile pentru precizie, senzitivitate și F1 obținute de implementările CNN și LSTM ale sistemului de recomandare propus comparate cu rezultatele obținute pe aceleași seturi de date de sistemul de recomandare descris în capitolul precedent și raportate în Florea *et al.* [29].

[2] https://deeplearning4j.org/about
[3] https://github.com/acflorea/deep-columbugus,mariana-triage

Tabela II.3.1: Netbeans - Precizia, senzitivitatea și F1 după 100 de epoci de antrenare cu loturi de 10 sau 25 de înregistrări comparate cu implementarea bazată pe SVM. Valorile optime sunt îngroșate

Setul de date	Netbeans				
Arhitectura	LSTM		**CNN**		SVM
Dimensiunea lotului	10	25	**10**	25	-
Precizia	0.86	0.85	**0.90**	0.88	0.89
Senzitivitatea	0.83	0.83	0.87	0.86	**0.88**
F1	0.84	0.84	**0.88**	0.87	**0.88**

Tabela II.3.2: Eclipse - Precizia, senzitivitatea și F1 după 100 de epoci de antrenare cu loturi de 10 sau 25 de înregistrări comparate cu implementarea bazată pe SVM. Valorile optime sunt îngroșate

Setul de date	Eclipse				
Arhitectura	LSTM		**CNN**		SVM
Dimensiunea lotului	10	25	**10**	**25**	-
Precizia	0.73	0.67	**0.80**	**0.80**	0.78
Senzitivitatea	0.70	0.68	0.75	0.75	**0.77**
F1	0.71	0.67	**0.77**	**0.77**	0.76

Tabela II.3.3: Mozilla - Precizia, senzitivitatea și F1 după 100 de epoci de antrenare cu loturi de 10 sau 25 de înregistrări comparate cu implementarea bazată pe SVM. Valorile optime sunt îngroșate

Setul de date	Mozilla				
Arhitectura	LSTM		**CNN**		SVM
Dimensiunea lotului	10	25	**10**	25	-
Precizia	0.59	0.62	**0.78**	0.77	0.77
Senzitivitatea	0.70	0.71	**0.75**	**0.75**	**0.75**
F1	0.64	0.66	**0.76**	0.75	0.73

Rezultatele obținute de versiunea bazată pe CNN a sistemului de recomandare prezentat sunt comparabile în ceea ce privește performanța clasificării cu cele obținute folosind SVM. Cu toate că LSTM are, în cea mai mare parte din cazuri, avantajul unui timp de antrenare mai redus, așa cum este prezentat în tabela II.3.4, performanțele obținute de clasificator sunt net sub cele ale arhitecturii CNN.

Din punct de vedere paralelism, ambele arhitecturi propuse conferă sistemului o scalabilitate ridicată, speedup-ul pentru 8 și 12 nuclee - versus 4 nuclee - fiind prezentate în tabela II.3.5.

Tabela II.3.4: Timpul de antrenare mediu per epocă, în secunde pentru 4 nuclee

Setul de date	Netbeans				Eclipse				Mozilla			
Arhitectura	LSTM		CNN		LSTM		CNN		LSTM		CNN	
Dimensiunea lotului	10	25	10	25	10	25	10	25	10	25	10	25
Timpul mediu de antrenare	408	212	442	175	1192	492	1411	525	3347	1028	3636	1579

Tabela II.3.5: Speedup-ul pentru 8 și 12 nuclee versus 4 nuclee

Setul de date	Netbeans				Eclipse				Mozilla			
Arhitectura	LSTM		CNN		LSTM		CNN		LSTM		CNN	
Dimensiunea lotului	10	25	10	25	10	25	10	25	10	25	10	25
Speedup 8 nuclee	1.83	2.93	2.85	1.84	2.02	2.14	2.34	2.16	2.84	1.66	2.01	1.82
Speedup 12 nuclee	2.53	4.06	4.31	4.12	2.52	2.69	3.96	3.67	5.23	3.12	2.94	3.06

II.3.3 Concluzii

Capitolul curent prezintă un sistem de recomandare pentru asignarea automată a erorilor în proiecte mari de tip software open source, bazat pe o arhitectură de tip învățare profundă. Atât CNN cât și LSTM sunt explorate ca arhitecturi posibile pentru nucleul sistemului de recomandare. Am propus o implementare paralelă, scalabilă, folosind un cluster de tip Apache Spark ceea ce permite atenuarea unei probleme majore în învățarea profundă și anume timpul lung de antrenare. Cu toate că implementarea bazată pe CNN obține rezultate echivalente cu cele obținute de implementarea paralelă folosind SVM, descrisă în capitolul precedent, SVM rămâne net superior din perspectiva timpului de antrenare necesar, mult mai redus. În ciuda superiorității SVM-ului, consider că arhitectura propusă deschide direcții noi de cercetare în zona optimizării vitezei de antrenare a rețelei (de exemplu prin folosirea de GPU-uri în loc de CPU-uri) cât și a optimizării clasificatorului prin identificarea de arhitecturi CNN mai performante.

Partea III

Optimizarea hiperparametrilor în învățarea automată

Această parte tratează problema optimizării hiperparametrilor în învățarea automată. Primul capitol este un capitol introductiv care prezintă problema optimizării hiperparametrilor și o parte din lucrările relevante în acest domeniu. Capitolul III.2 propune un criteriu dinamic de oprire automată pentru algoritmul de tip RS (Random Search - Căutare aleatoare) cu aplicare directă în cazul optimizării de hiperparametri pentru modelele de tip învățare automată. Criteriul propus duce la reducerea semnificativă a numărului de încercări necesare algoritmului RS, fără a influența negativ rezultatul obținut în urma optimizării. Este prezentată, de asemenea, o variantă paralelă pentru algoritmul introdus. Probabilitatea de a obține un rezultat optim după un număr redus de încercări crește cu numărul de nuclee folosite de acestă implementare paralelă. Capitolul III.3 introduce o variantă modificată de RS. Algoritmul propus se bazează pe asignarea de probabilități de schimbare pentru fiecare din hiperparametrii țintă. La fiecare iterație, în funcție de aceste probabilități, pentru fiecare hiperparametru, fie este generată o valoare nouă, fie este folosită cea mai bună valoare obținută până atunci. Am demonstrat teoretic că probabilitatea ca algoritmul propus să identifice valoarea optimă căutata, după un număr de încercări dat, este mai mare decât în cazul RS. Am testat algoritmul în contextul optimizării unei variații a funcției Grievank cât și pentru optimizarea hiperparametrilor unei rețele neurale de tip CNN. În ambele cazuri rezultatele obținute au fost superioare celor obținute de alte tehnici de optimizare.

Capitolul III.1
Stadiul actual al cercetării

Capitolul curent descrie pe scurt problema optimizării hiperparametrilor în învățarea automată cu accent pe importanța problemei atât în contextul SVM cât și în contextul unei arhitecturi mai complexe, din zona învățării profunde, și anume CNN. Sunt prezentate câteva din lucrările semnificative în domeniul optimizării hiperparametrilor, în special în zona căutării aleatorii (RS).

Marea majoritate a algoritmilor de tip învățare automată sunt caracterizați de două seturi diferite de parametri: parametrii de antrenare și meta-parametrii (mai poartă numele de *hiperparametri*). Identificarea, în cazul hiperparametrilor, a valorilor care produc rezultate optime (sau aproape de optim) este esențială pentru a garanta o bună generalizare a modelului obținut. Alegerea acestor valori se face prin antrenarea repetată a modelelor generate de combinații diferite de valori pentru hiperparametri. O singura secvență de antrenare plus evaluare poartă numele de *încercare(trial)*. Antrenarea unui model este, în general, o operație care necesită un consum semnificativ de resurse. În plus, numărul de încercări necesare crește exponențial cu numărul de hiperparametri caracteristici modelului. Este așadar importantă reducerea numărului de încercări necesare pentru a determina combinația optimă de valori pentru acești hiperparametri.

În cazul algoritmilor clasici din zona învățării automate, alegerea unui set de valori optim pentru acești hiperparametri are o influență semnificativă asupra performanțelor modelului. În cazul SVM, de exemplu, modelul obținut depinde de mai mulți hiperparametri și este foarte sensibil la schimbări în valoarea acestora [21]. Alegerea nucleului poate avea o influență dramatică asupra performanței clasificatorului [18]. Parametrul de regularizare (C), care controlează echilibrul dintre maximizarea marginii de separare și minimizarea erorii de clasificare este de asemenea foarte important deoarece, în cazul claselor non-separabile, algoritmul trebuie să permită erori de clasificare. În cazul specific al unui nucleu polinomial, o alegere greșită a gradului polinomului poate duce ușor la over-fitting [11]. Recent, interesul în zona optimizării de hiperparametri este în continuă creștere, în special datorită creșterii în popularitate a modelelor de tip învățare profundă, modele care pun o presiune deosebită asupra tehnicior existente din cauza numărului foarte

mare de hiperparametri implicați cât și a timpului mare de antrenare necesar pentru aceste arhitecturi. În cazul CNN, de exemplu, numărul de hiperparametri caracteristici rețelei poate ușor exploda (numărul de straturi de convoluție, numărul de straturi de pooling, numărul de convoluții în fiecare strat, numărul de straturi dense și numărul de neuroni pentru fiecare strat în parte, funcțiile de activare pentru fiecare strat, valoarea pentru dropout etc.) Valori diferite pentru acești hiperparametri duc la rețele cu structuri diferite și implicit performanțe diferite, cum ar fi LeNet-5 [50], AlexNet [48], ZFNet [88], GoogleNet [81], VGGNet [76], RESNet [37] etc.

Următoarele două capitole descriu două abordări diferite asupra problemei optimizării hiperparametrilor. Prima abordare propune un criteriu dinamic pentru oprirea timpurie a optimizării în cazul căutării aleatoare (RS) iar cea de-a doua, o variantă optimizată a algoritmului de tip RS și anume căutarea aleatorie ponderată (Weighted Random Search - WRS) unde, pentru încercările efectuate, valorile hiperparametrilor sunt generate cu o anumită probabilitate, specifică fiecărui hiperparametru în parte. Această strategie duce la o probabilitate mai bună de a atinge optimul comparativ cu RS.

De-a lungul timpului au fost dezvoltate diverse metode pentru optimizarea hiperparametrilor, pornind de la cele mai simple, cum ar fi Grid Search (GS) sau ajustarea manuală [38, 51, 77][1] și până la tehnici mult mai elaborate ca Nelder-Mead [2, 59], Simulated Annealing [45], algoritmi evolutivi [35], metode Bayes-iene [82] etc.

În prezent interesul în zona optimizării hiperparametrilor oscilează între introducerea unor tehnici din ce în ce mai sofisticate (Sequential Model-Based Global Optimization [42], învățare prin recompensă - reinforcement learning [89, 90] etc.) și diverse încercări de optimizare a tehnicilor existente.

Căutarea Aleatoare (RS) face parte din categoria algoritmilor simpli [9, 10]. În limitele aceluiași buget computațional, RS obține, în general, rezultate mai bune decât GS sau tehnici mai complicate de optimizare a hiperparametrilor [9]. În special când numărul de dimensiuni este mare, resursele computaționale necesare pentru RS sunt semnificativ mai reduse decât în cazul GS [52]. RS constă în generarea de valori independente, conform cu o anumită funcție de distribuție, pentru fiecare din hiperparametri. Fiecare încercare poate fi așadar evaluată independent de celelalte, ceea ce face din RS un candidat ideal pentru paralelizare.

Câteva încercări recente de a optimiza RS sunt date de Hyperband, Li's *et al.* [53], care accelerează RS prin alocarea adaptivă a resurselor și prin oprire timpurie

[1] https://github.com/jaak-s/nips2014-survey - 82 din 86 de lucrări legate de optimizarea hiperparametrilor prezentate la NIPS 2014 folosesc GS.

a algoritmului, Domhan *et al.* [24] unde este prezentat un model probabilistic pentru a opri anticipat antrenarea în cazul încercărilor cu potențial sub-optim, Florea *et al.* [26], unde am introdus un criteriu dinamic pentru oprirea timpurie, reducând numărul de încercări necesare fără a reduce performanțele modelului obținut sau Florea *et al.* [27] unde am introdus o variație a RS care, în cadrul aceluiași buget computațional obține rezultate net superioare pentru optimizarea hiperparametrilor unei rețele de tip CNN.

Capitolul III.2

Un criteriu dinamic pentru oprirea timpurie în cazul căutării aleatorii

Acest capitol este o extensie a rezultatelor prezentate în cadrul conferinței ”IFIP International Conference on Artificial Intelligence Applications and Innovations” în 2018 și publicate în Springer, ”IFIP Advances in Information and Communication Technology” volumul 519 cu titlul: ”A Dynamic Early Stopping Criterion for Random Search in SVM Hyperparameter Optimization” [26]. Capitolul propune o metodă de accelerare a algoritmului de tip Căutare Aleatoare (RS) în contextul optimizării de hiperparametri. Nucleul acestei optimizări constă în introducerea unui criteriu de oprire timpurie, calculat dinamic. Algoritmul propus este implementat în paralel și obține o scalabilitate ridicată. Am testat algoritmul propus în cazul optimizării hiperparametrilor pentru un model de tip SVM, pe șase seturi de date uzuale. Conform testelor efectuate, algoritmul introdus accelerează semnificativ optimizarea de tip RS fără a avea o influență negativa asupra performanțelor clasificatorului.

Capitolul continuă după cum urmează: inițial este descris algoritmul propus și condiția dinamică de oprire, cu accent pe caracterul paralel al algoritmului. Secțiunea III.2.2 prezintă apoi rezultatele experimentale și capitolul se încheie cu secțiunea III.2.3 unde sunt prezentate concluziile.

III.2.1 Algoritmul propus - proprietăți probabilistice

Secțiunea curentă descrie algoritmul propus atât din perspectiva unei implementări secvențiale cât și a unei implementări paralele. Este descris criteriul de oprire timpurie introdus și modul în care acesta a fost dedus.

În cazul RS, un generator g are ca sarcină furnizarea de valori conforme cu funcția de distribuție aferentă fiecărui hiperparametru în parte, algoritmul continuând întotdeauna până la epuizarea unui număr maxim de N încercări. Valoarea optimă raportată este cea mai bună - în conformitate cu criteriul de performanță ales - din cele N obținute. Scopul criteriului de oprire timpurie, introdus în continuare, este

acela de a reduce numărul de încercări la mai puțin de N, fără un efect semnificativ asupra valorii optime raportate.

Modul în care acest criteriu de oprire este introdus într-un algoritm de optimizare de hiperparametri este detaliat în Algoritmul 1 - în acest caz este descris un algoritm de maximizare. Acest algoritm este un algoritm de optimizare în două etape. Inițial se iterează pentru un număr de pași predefinit, n, $n \ll N$, și se obține combinația de hiperparametri care determină un optim temporar, tmp_opt. A doua etapă constă în încercarea de a identifica primul set de valori pentru hiperparametri care generează o valoare mai bună decât tmp_opt. Rezultatul optim raportat, opt, este fie primul rezultat mai bun decât tmp_opt, obținut în etapa a doua, fie chiar tmp_opt dacă niciun rezultat mai bun nu este obținut în a doua etapă și numărul maxim de încercări N este atins.

Am notat cu F funcția obiectiv. De cele mai multe ori nu este cunoscută expresia analitică a funcției dar aceasta poate fi calculată pentru un punct dat. În cazul curent calculul valorilor pentru această funcție constă într-o sesiune de tip antrenare-evaluare a modelului propus pentru optimizare. Datele de intrare pentru Algoritmul 1 constau așadar într-un mod de a calcula funcția F, numărul maxim de iterații (încercări) N, numărul de încercări corespunzătoare primei faze, n, și funcția de distribuție pentru fiecare hiperparametru, $P_i(x), i = 1, \ldots, d$, unde d este numărul total de hiperparametri. Rezultatul algoritmului de optimizare constă în cea mai bună valoare.

Dacă g_opt este cel mai bun rezultat posibil obținut prin testarea tuturor celor N seturi de valori atunci, implicit, acesta este și rezultatul raportat de implementarea standard de RS. Am demonstrat că valoarea optimă a lui n astfel încât probabilitatea de a identifica $opt = g_opt$ după cel mult m iterații, unde $n < m < N$ este dată de:

$$n = \frac{m}{e} \tag{III.2.1}$$

și, în acest caz, valoarea acestei probabilități este dată re relația III.2.2:

$$P(E) = P_m(opt = g_opt) = \frac{n}{N} \sum_{i=n}^{m-1} \frac{1}{i} \tag{III.2.2}$$

Alegând o valoare a lui n mai mare decât m/e este mărită probabilitatea ca Algoritmul 1 să identifice g_opt dar cu dezavantajul creșterii numărului de iterații.

Algorithm 1 Optimizare cu criteriu de oprire timpurie

Intrare: F; N; n; $P_i(x), i = 1, \ldots, d$
Ieşire: $index$, X, opt

// Faza 1
Iniţializează tmp_opt cu $-math.MaxFloat64$
for $k = 1$ to $n \ll N$ **do**
  Generează X^k
  Calculează $F(X^k)$
  if $F(X^k) \geq tmp_opt$ **then**
    $tmp_opt = F(X^k)$
  end if
end for
// Faza 2 - oprire timpurie
$opt = tmp_opt$
for $k = n + 1$ to N **do**
  Generează X^k
  Calculează $F(X^k)$
  if $F(X^k) \geq tmp_opt$ **then**
    $opt = F(X^k)$
    BREAK
  end if
end for
return k, X^k, opt

Am propus de asemenea o implementare paralelă care măreşte probabilitatea de a obţine g_opt după un număr semnificativ redus de încercări. Secţiunea următoare detaliază implementarea paralelă propusă.

III.2.1.1 Execuţia paralelă

Calculul efectiv al valorii funcţiei F este, de cele mai multe ori, o operaţie complexă şi consumatoare de resurse, de aceea devine importantă problema paralelizării algoritmului propus. Secţiunea curentă prezintă astfel o modalitate posibilă de a paraleliza Algoritmul 1.

O versiune paralelă a Algoritmului 1 poate fi implementată astfel:

- Se împart sarcinile între W workeri (aceşti workeri pot fi orice, de la fire de execuţie, thread-uri ale sistemului de operare, nuclee ale unui procesor sau

chiar servere diferite). Pentru implementarea curentă am decis să folosesc suportul oferit de limbajul GOLANG [64] care oferă posibilitatea de a paraleliza orice implementare cu ajutorul goroutine-lor[1]. Aceste goroutine sunt în esență un fel fire de execuție gestionate de limbaj în sine.

- Fiecare worker w execută maximum N/W iterații, folosind același criteriu de oprire. În acest caz numărul de iterații aferent fazei 1 este, pentru fiecare worker, $n_w = n/W$. Atunci, pentru o valoare a lui $m = N/2$ fiecare nod worker va termina după aproximativ $N/(2W)$ iterații, N/W fiind cazul cel mai defavorabil. La terminarea execuției, fiecare worker va raporta așadar un optim local.
- Un nod de tip manager adună rezultatele tuturor worker-ilor și alege cel mai bun optim local.

Algoritmul 2 prezintă implementarea descrisă anterior. Oricare din următoarele strategii de generare în paralel a numerelor pseudo-aleatoare [69] poate fi aleasă: Manager-Worker (MW), Sequence Splitting (SeqS), Leapfrog (LF) și Parametrization (P).

Algorithm 2 Optimizare cu criteriu de oprire timpurie - implementarea paralelă

Intrare: W, F; N; $P_i(x), i = 1, \ldots, d$
Ieșire: $index$, X, opt
// Faza 1- Distribuie execuția către workeri
1: **for** $w = 1$ to W **do**
2: Lansează worker_w(F; N/W; $P_i(x), i = 1, \ldots, d$) *// Fiecare worker rulează Alg. 1 cu* $n = N/(eW)$
3: **end for**
// Faza 2 - Colectare rezultate
4: **for** $w = 1$ to W **do**
5: (k_w, X^k_w, opt_w) = rezultat worker_w
6: **end for**
7: Calculează opt ca Max(opt_w); k, X^k corespund lui opt
8: **return** k, X^k, opt

Plecând de la relația (III.2.2), probabilitatea ca oricare dintre workeri să identifice cel mai bun rezultat din cele N posibile este:

[1] https://tour.golang.org/concurrency/1

$$P_W(E) = \frac{n}{N} \sum_{i=n/W}^{m/W-1} \frac{1}{i} \quad \text{(III.2.3)}$$

și am demonstrat că $P_W(E) > P(E)$:

Astfel se obține că implementarea paralelă propusă are o probabilitate mai mare de a identifica g_opt comparativ cu implementarea secvențială. Mai mult decât atât, cu cât numărul de workeri crește cu atât $P_W(E)$ crește.

Detalii de implementare

Am folosit limbajul de programare GO[2] pentru a implementa algoritmul propus datorită suportului său implicit pentru programare concurentă cât și datorită vitezei impresionante conferită de acest limbaj în comparație cu altele[3]. Codul este disponibil public pe GitHub[4]. Implementarea paralelă se bazează pe goroutine și canale - componente standard oferite de limbajul GO.

Pachetul de optimizare creat acceptă următorii parametri:

- *fileName* - Locația fișierului care conține datele de antrenare. Algoritmul acceptă date în format LIBSVM [17].
- *noOfExperiments* - Numărul de experimente independente care vor fi executate.
- *maxAttempts* - Numărul maxim de încercări (N) pentru un experiment.
- *alg* - Algoritmul utilizat pentru generarea în paralel a numerelor aleatoare: MW pentru ManagerWorker, SeqSplit pentru Sequence Split, Leapfrog sau Parametrization. SeqSplit este valoarea implicită.
- *w* - Numărul de workeri, valoarea implicită este 8.

III.2.2 Experimente

Secțiunea curentă prezintă detalii asupra experimentelor efectuate și a seturilor de date folosite pentru a testa algoritmul propus.

[2] https://golang.org/
[3] http://www.techempower.com/benchmarks/
[4] https://github.com/acflorea/goptim

Am folosit metoda propusă pentru a optimiza cinci hiperparametri aferenți unui clasificator de tip SVM [21] și anume:

- tipul nucleului (RBF, Polynomial sau Linear, fiecare valoare aleasă cu o probabilitate egală)
- γ - generată conform unei distribuții exponențiale cu $\lambda = 10$
- *parametrul de regularizare* (*C*) - generat conform unei distribuții exponențiale cu $\lambda = 10$
- *degree* - gradul polinomului folosit în cazul nucleului polinomial, generat cu o probabilitate egală din mulțimea $\{2, 3, 4, 5\}$
- *coef0* - termenul liber folosit în cazul nucleului polinomial, generat uniform în $[0, 1]$

Am testat algoritmul pe șase din cele mai populare seturi de date disponibile pe site-ul UCI Machine Learning Repository [5]: Adult (a1a), Adult (a6a), Breast Cancer, Diabetes, Iris și Wine. Adult (a1a) și Adult (a6a) sunt variații ale aceluiași set de date dar cu dimensiuni diferite (al doilea este aproximativ de șase ori mai mare decât primul). Am folosit acuratețea [79] clasificării ca măsură a performației modelului, calculată folosind validarea încrucișată cu zece sub-seturi de date (ten fold cross-validation). Am comparat rezultatele obținute, atât ca acurateț cât și ca număr de încercări, cu implementări standard ai unor algoritmi pentru optimizarea de hiperparametri, implementări oferite de biblioteca Optunity și anume: GS, RS, Particle Swarm și Nelder-Mead cât și cu implementarea SVM din Weka, de data aceasta folosind valorile propuse implicit pentru hiperparametrii țintă. În timpul testelor, am rulat algoritmul de optimizare folosind $W = 8$ și $N = 250$, deci o valoare a lui $n = 92$. Am rulat, de asemenea, algoritmii de optimizare din Optunity pentru un număr de maxim 250 încercări.

III.2.2.1 Evaluarea modelului obținut

Secțiunile următoare prezintă evaluarea algoritmului propus atât din perspectiva performațelor (acuratețea) modelului obținut cât și din punct de vedere al bugetului computațional necesar. Este analizată, de asemenea, scalabilitatea implementării paralele.

Tabela III.2.1 prezintă rezultatele aplicării Algoritmului 2 pentru fiecare din cele patru strategii de generare în paralel a numerelor pseudo-aleatoare (Manager-Worker, Sequence Split, Leapfrog și Parametrization). Tabela III.2.2 prezintă rezultatele

[5] http://archive.ics.uci.edu/ml/index.php

obținute folosind Optunity (RS, GS, Particle Swarm și Nelder-Mead) cât și rezultatele obținute de implementarea SVM din Weka, cu valorile implicite pentru hiperparametri ($C = 1.0$, $\gamma = 1/number_of_features$, $degree = 3$, $coef0 = 0.0$). Cele mai bune rezultate sunt marcate îngroșat.

Tabela III.2.1: Acuratețea și numărul de încercări pentru Algoritmul 2 folosind cele patru strategii de generare în paralel a numerelor pseudo-aleatoare (MW, SeqS, LF, P)

	Manager Worker		Sequence Split		Leapfrog		Parametrization	
Setul de date	Acuratețea	Încercări	Acuratețea	Încercări	Acuratețea	Încercări	Acuratețea	Încercări
Adult (a1a)	**0.837**	164	0.836	194	**0.837**	194	**0.837**	**148**
Adult (a6a)	0.844	169	0.844	**133**	0.844	203	**0.845**	138
Cancer	**0.975**	172	**0.975**	**167**	**0.975**	214	**0.975**	187
Diabetes	0.776	223	0.776	203	**0.780**	220	0.779	**130**
Iris	0.973	224	**0.980**	**156**	**0.980**	189	0.973	158
Wine	**0.989**	194	**0.989**	215	**0.989**	**162**	**0.989**	177
Media	0.899	191	0.900	**178**	**0.901**	197	0.900	156.334
Abaterea standard	0.091	27.188	0.092	31.241	0.091	20.746		22.223

Tabela III.2.2: Acuratețea și numărul de încercări pentru algoritmii de optimizare de hiperparametri implementați în Optunity (RS, GS, Particle Swarm și Nelder-Mead) și pentru Weka's SVM

	Optunity RS	Optunity GS	Optunity PSO	Optunity NM		Weka RBF	Weka Poli	Weka Lineal
Setul de date	Acuratețea	Acuratețea	Acuratețea	Acuratețea	Încercări	Acuratețea	Acuratețea	Acuratețea
Adult (a1a)	0.837	0.835	0.838	0.833	108	0.828	**0.839**	0.754
Adult (a6a)	0.844	0.844	**0.845**	0.843	142	0.838	0.760	0.760
Cancer	0.975	0.975	**0.975**	0.968	17	0.969	0.974	0.969
Diabetes	**0.777**	**0.777**	0.776	0.651	6	0.775	0.686	**0.777**
Iris	0.980	**0.987**	0.967	0.940	6	0.953	0.727	0.960
Wine	**0.989**	0.984	**0.989**	0.956	62	0.983	0.404	0.972
Media	**0.900**	**0.900**	0.898	0.865	56.834	0.891	0.732	0.865
Abaterea standard	0.092	0.092	0.090	0.120	57.711	0.088	0.190	0.112

Fiecare algoritm de optimizare are ca rezultat un anumit model SVM. Performanța fiecărui algoritm este așadar pe de o parte performanța clasificării (acuratețea) modelului rezultat și, pe de altă parte, numărul de încercări (bugetul computațional) necesar optimizării. Deoarece sunt comparate mai multe tehnici pe mai multe seturi de date sunt necesare teste statistice adiționale pentru a evalua performanța acestor algoritmi [23]. Am calculat așadar valorile statistice standard Friedman [30] și Iman-Davenport [43]. Aceste valori statistice iau în calcul ordinea algoritmilor pentru fiecare set de date în parte, în funcție de performanța obținută pentru setul de date respectiv.

Cu 11 algoritmi și șase seturi de date, valoarea lui F_F urmează o distribuție F, cu $11-1 = 10$ și, respectiv, $(11-1)\cdot(6-1) = 50$ de grade de libertate [23]. Valoarea critică pentru $F(10, 50)$ și $\alpha = 0.05$ este 2.03 deci ipoteza nulă (conform căreia toți

cei 11 algoritmi sunt identici ca performanță) poate fi respinsă (F_F pentru algoritmii testați este $6.04 > 2.03$). Este evident așadar că cei 11 algoritmi nu sunt identici ca performanță.

Valoarea diferenței critice [23, 60], la un nivel de semnificație $\alpha = 0.05$, este $CD_{0.05} = 6.163$. Această valoare exclude categoric algoritmul Nelder-Mead care este semnificativ mai puțin performant decât implementarea de tip Parametrization a Algoritmului 2. La un nivel de semnificație $\alpha = 0.1$, diferența critică este $CD_{0.1} = 5.701$ și se poate observa că Optunity - NM este semnificativ mai puțin performant decât GO-LF și GO-P și că GO-P este semnificativ mai performant decât Optunity - NM și Weka - RBF. Se pot distinge așadar trei categorii de algoritmi din punct de vedere a performanței obținute de clasificatorul rezultat, de la cel mai bun la cel mai rău:

- GO Parametrized
- Restul algoritmilor (GO-LF, GO-SeqS, GO-MW, Optunity RS, GS și PSO, implementarea SVM Weka)
- Optunity Nelder-Mead

În concluzie, Algoritmul 2 duce la obținerea unui model cu o performanță în clasificare cel puțin egală cu cea a algoritmilor standard.

III.2.2.2 Eficiența criteriului de oprire

Plecând de la rezultatele prezentate anterior, exclud Nelder-Mead (din cauza performanței modelului obținut semnificativ mai scăzută în comparație cu toate celelalte modele) din comparația numărului de încercări. Exclud, de asemenea, implementările Weka deoarece, în cazul lor, nu este vorba de o optimizare de hiperparametri propriu-zisă (există o singură încercare, cu valorile implicite). Tabela III.2.3 prezintă diferența între pozițiile fiecărui algoritm într-un clasament bazat pe numărul de încercări efectuate (cea mai bună poziție corespunde celor mai puține încercări necesare - deci bugetului computațional necesar minim).

Am rulat un nou test Friedman și am obținut $\chi^2_F = 28.554$ și $F_F = 19.173$. Valoarea critică, în acest caz, $F(6, 30)$ ($7 - 1$ și respectiv $(7 - 1)(6 - 1)$) este 2.420. Aceasta înseamnă că ipoteza nulă (care prevede că toți algoritmii sunt echivalenți ca performanță) poate fi exclusă ($19.173 > 2.420$).

Valoarea diferenței critice, pentru o semnificație de 0.05 este în acest caz: $CD_{0.05} = 3.678$. Valorile mai mari decât $CD_{0.05}$ sunt marcate cu caractere îngroșate în tabela III.2.3. Se pot astfel identifica două grupuri de algoritmi. Primul grup (GO - SeqS și

Tabela III.2.3: Diferența între pozițiile ocupate de fiecare algoritm - în funcție de numărul de încercări necesare

	GO MW	GO SeqS	GO LF	GO P	Optunity RS	Optunity GS	Optunity PSO
GO MW	-	-0.917	0.084	-1.167	3	3	3
GO SeqS		-	1	-0.25	**3.917**	**3.917**	**3.917**
GO LF			-	-1.25	2.917	2.917	2.917
GO P				-	**4.167**	**4.167**	**4.167**
Optunity RS					-	0	0
Optunity GS						-	0
Optunity PSO							-

Tabela III.2.4: Performanța raportată la numărul de încercări necesare pentru GO - MW, GO - P, GO - SeqS și GO - LF în raport cu Optunity - RS. Date caracteristice testului Holm

	i	$z = (R0 - Ri)/SE$	p	α/i
GO P	1	3.341	0.00084	0.0084
GO SeqS	2	3.140	0.00043	0.01
GO MW	3	2.405	0.00808	0.0125
GO LF	4	2.339	0.009668	0.0167

GO - P) obține, din punct de vedere statistic, performanțe mai bune decât grupul format din Optunity - GS, Optunity - RS și Optunity - PSO. Nu este evident cărui din cele două grupuri aparțin algoritmii GO - MW și GO - LF. O explicație posibilă pentru rezultatele mai bune obținute folosind Sequence Split și Parametrization comparativ cu celelalte două tehnici (MW și LF), poate fi legată de o calitate mai bună a valorilor pseudo-aleatoare generate folosind aceste tehnici de paralelizare a generatorului. Totuși, deoarece numărul de valori necesare pentru testele efectuate este relativ mic, cel mai probabil diferența este pur întâmplătoare.

În final, deoarece scopul principal al Algoritmului 2 este obținerea unei variante mai bune de RS, compar metoda prezentată direct cu implementarea RS din Optunity folosind testul Holm [40]. Eroarea standard pentru experimentul efectuat este $SE = \sqrt{k(k+1)/(6N)} = 1.247$ (și aici, N este numărul de seturi de date folosite și k numărul total de algoritmi). Tabela III.2.4 conține rezultatele testului de respingere Holm.

Testul Holm respinge așadar toate cele patru ipoteze ($p < \alpha/i$), ceea ce duce la concluzia că toate cele patru versiuni (tipuri de generator) propuse pentru Algoritmul 2 sunt semnificativ mai eficiente în ceea ce privește numărul de încercări

efectuate comparativ cu implementarea RS standard.

III.2.2.3 Scalabilitatea implementării paralele

Pe lângă acurateţe şi număr de încercări necesare, fiind vorba de o implementare paralelă, este important de testat această implementare şi din perspectiva scalabilităţii. Am testat Algoritmul 2 pe un număr din ce în ce mai mare de nuclee de calcul şi am calculat astfel speedup-ul (raportul dintre timpul necesar unei execuţii secvenţiale şi timpul necesar execuţiei paralele) ca măsură a scalabilităţii. Valorile obţinute sunt prezentate în tabela III.2.5. Pentru toate seturile de date, valorile cresc proporţional cu numărul de nuclee ceea ce demonstrează o bună scalabilitate a algoritmului propus.

Tabela III.2.5: Speedul-ul pentru Algoritmul 2 raportat la numărul de nuclee

Setul de date/Numărul de nuclee	2	3	4	6	8
Adult(a1a)	1.37	2.96	3.48	4.13	4.52
Adult (a6a)	1.97	2.71	3.02	3.35	3.70
Cancer	1.98	2.91	3.53	3.91	4.09
Diabetes	1.86	2.72	3.34	3.70	3.88
Iris	1.94	2.75	3.10	3.43	3.54
Wine	1.99	2.81	3.28	3.96	4.11
Average	1.85	2.81	3.29	3.75	3.97

III.2.3 Concluzii

Capitolul curent a introdus un criteriu dinamic pentru oprirea timpurie a algoritmului de tip căutare aleatoare (RS) destinat optimizării de hiperparametri. Este propusă, de asemenea, o variantă paralelă a unui algoritm bazat pe acest criteriu. Am demonstrat teoretic că probabilitatea de a identifica optimul într-un număr de încercări redus comparativ cu RS creşte în cazul algoritmului paralel, comparativ cu versiunea lui secvenţială.

În contextul optimizării unui clasificator de tip SVM, pe şase din cele mai folosite seturi de date pentru o astfel de problemă, din punct de vedere al acurateţei modelului generat, am obţinut rezultate similare cu tehnicile standard existente în domeniul optimizării de hiperparametri. Am folosit patru tehnici de generare paralelă de numere pseudo-aleatoare. Cu toate cele patru tehnici, algoritmul

termină într-un număr de încercări semnificativ mai mic decât varianta standard de RS, ceea ce duce la o reducere semnificativă a bugetului computațional necesar optimizării.

Algoritmul propus deschide noi perspective de cercetare în zona optimizării de hiperparametri pentru alți algoritmi de tip învățare automată, în special când spațiul de căutare are un număr mare de dimensiuni (număr mare de hiperparametri) și bugetul computațional necesar constituie un impediment major.

Implementarea algoritmului este suficient de flexibilă pentru a putea fi adaptată oricărei optimizări de tip ”gradient-free”.

Capitolul III.3

Căutarea aleatorie ponderată

Rezultatele prezentate în acest capitol au la baza articolul cu titlul ”Weighted Random Search for Hyperparameter Optimization” [27], publicat în International Journal of Computers Communications & Control (IJCCC) Nr. 14(2), Aprilie 2019 - Factor de impact în JCR2017 (Clarivate Analytics, SCI Expanded, ISI Web of Science): IF=1.29. Capitolul prezent propune o variantă a metodei de tip căutare aleatoare (RS) în contextul optimizării de hiperparametri pentru algoritmii de tip învățare automată. Spre deosebire de implementarea standard RS, care, la fiecare iterație, generează valori noi pentru fiecare hiperparametru, algoritmul propus generează valori noi cu o anumită probabilitate de schimbare p și folosește cele mai bune valori identificate până atunci cu probabilitatea $1 - p$. Intuitiv, o valoare care deja a dus la obținerea unui rezultat promițător este un bun candidat pentru a fi testată în combinații noi de valori pentru ceilalți hiperparametri.

Am denumit algoritmul propus Căutare Aleatorie Ponderată (Weighted Random Search - WRS). Pentru același număr de încercări (același buget computațional), WRS obține rezultate seminificativ mai bune decât RS. Am testat algoritmul pe o variație a unei funcții cunoscute în domeniul optimizării - funcția Grievank [34] - cât și în contextul optimizării de hiperparametri pentru arhitectura unei rețele neurale de tip CNN. Am obținut de asemenea rezultate teoretice care justifică performanța superioară a WRS în raport cu RS.

Capitolul debutează cu descrierea algoritmului propus, apoi sunt prezentate aspectele teoretice legate de convergenta WRS și o comparație cu RS din perspectiva probabilității de a identifica optimul căutat. În continuare sunt prezentate rezultatele obținute de WRS pentru optimizarea unei variații a funcției Grievank și apoi pentru optimizarea hiperparametrilor în cazul unei rețele neurale de tip CNN. Ultima secțiune prezintă concluziile.

III.3.1 Metoda WRS

Acestă secțiune debutează cu o descriere intuitivă a algoritmului WRS, algoritm care reprezintă nucleul capitolului curent. Ulterior sunt prezentate detaliile tehnice ale algoritmului.

Tehnica RS clasică [9] generează o nouă valoare pentru o variabilă pseudo-aleatoare multi-dimensională la fiecare iterație k, cu valori noi pentru fiecare din dimensiuni - $X^k = \{x_i^k\}, i = 1, \ldots, d$ - unde x_i este generată în conformitate cu o funcție de distribuție $P_i(x), i = 1, \ldots, d$ iar d este numărul de dimensiuni.

WRS este o variantă îmbunătățită de RS, destinată optimizării de hiperparametri. Algoritmul atribuie câte o probabilitate de schimbare $p_i, i = 1, \ldots, d$ fiecărei dimensiuni și, după un anumit număr de iterații k_i, pentru fiecare dimensiune i, în loc să genereze o valoare nouă, generează această valoare cu probabilitatea p_i. Cu probabilitatea $1 - p_i$ este folosită cea mai bună valoare a lui x_i identificată până la pasul respectiv.

WRS constă în două etape. În prima fază rulează RS pentru un număr de încercări (iterații) predefinit. Acestă etapă permite:

- Identificarea celei mai bune combinații de valori până la momentul curent
- Obținerea unor date de intrare suficiente cu privire la importanța fiecărei dimensiuni în procesul de optimizare.

Cea de-a doua fază ia în calcul probabilitățile de schimbare și generează seturi noi de valori candidat în conformitate cu acestea. Între cele două etape, este executată o instanță de fANOVA [41] pentru a determina importanța fiecărei dimensiuni în variația funcției obiectiv. Intuitiv, cea mai importantă dimensiune (dimensiunea care este responsabilă pentru cea mai mare variație a funcției obiectiv) este aceea ale cărei valori ar trebui să se schimbe cel mai des, pentru a acoperi cât mai mult din variația de care este responsabilă. Pentru o dimensiune responsabilă de o mică parte din variația funcției obiectiv, ar putea fi mai eficient de păstrat un optim temporar, odată ce această valoare a fost identificată.

O iterație a algoritmului WRS, aplicat în cazul maximizării unei funcții, este descrisă în Algoritmul 3 iar metoda în ansamblu este prezentată în Algoritmul 4. F este funcția obiectiv, de cele mai multe ori forma analitică a funcției nu este cunoscută și valoarea lui $F(X)$ trebuie calculată la fiecare iterație, X^k este cel mai bun argument la iterația k iar N este numărul total de iterații (în contextul optimizării de hiperparametri fiecare iterație corespunde unei încercări, cei doi termeni sunt așadar inter-schimbabili).

Pentru fiecare iterație a Algoritmului 4, cel puțin o dimensiune se schimbă de aceea, cel puțin una din valorile probabilităților p_i va fi egală cu unu. Pentru celelalte valori ale probabilităților de schimbare, orice valoare în $(0, 1]$ este validă. Dacă toate valorile sunt unu atunci WRS devine identic cu RS.

Algorithm 3 O iterație WRS - Maximizarea funcției obiectiv

Intrare: F; (X^k, $F(X^k)$); $p_i, k_i, P_i(x), i = 1, \ldots, d$
Ieșire: $(X^{k+1}, F(X^{k+1}))$

Generează aleator p, uniform în (0,1)
for $i = 1$ to d **do**
  if ($p_i \geq p$ or $k \leq k_i$) **then**
    // sau este satisfăcută condiția de probabilitate
    // sau sunt necesare mai multe valori
    Generează x_i^{k+1} conform cu $P_i(x)$
  else
    $x_i^{k+1} = x_i^k$
  end if
end for
// De obicei aceasta este etapa cea mai costisitoare
Calculează $F(X^{k+1})$
if $F(X^{k+1}) \geq F(X^k)$ **then**
  return $(X^{k+1}, F(X^{k+1}))$
else
  return $(X^k, F(X^k))$
end if

Algoritmul 4 coordonează secvența de pași descrisă și apelează Algoritmul 3 într-o buclă, până când numărul maxim de încercări N este atins.

III.3.2 Aspecte teoretice, convergența algoritmului

Secțiunea curentă analizează convergența Algoritmului 4 din perspectivă teoretică și compară Algoritmul 4 cu RS. Similar cu GS și RS, și în cazul WRS, fac presupunerea că variabilele funcției obiectiv nu sunt corelate statistic. Pentru a face explicațiile mai intuitive, inițial este discutat cazul bi-dimensional și apoi rezultatele obținute sunt generalizate pentru cazul multi-dimensional. În încheierea secțiunii este introdusă, de asemenea, o propunere de alegere a unui "set bun de valori" pentru p_i și $k_i, i = 1, \ldots, d$ (folosite la pasul 4 și 5 în Algoritmul 4). Prin $n \geq 1$ am definit numărul de iterații atât în cazul WRS cât și în cazul RS.

Algorithm 4 WRS - Maximizarea funcției obiectiv

Intrare: F; N; $P_i(x), i = 1, \ldots, d$
Ieșire: $(X^N, F(X^N))$
// Faza 1 - Execuție RS
1: **for** $k = 1$ to $N_0 < N$ **do**
2: Execută o iterație RS, calculează $(X^k, F(X^k))$
3: **end for**
// Faza intermediară - determină valorile de intrare pentru WRS
4: Determină probabilitățile de schimbare $p_i, i = 1, \ldots, d$
5: Determină numărul minim de valori necesare $k_i, i = 1, \ldots, d$
// Faza 2 - Execuție WRS
6: **for** $k = N_0 + 1$ to N **do**
7: Execută o iterație WRS descrisă în Algoritmul 3, calculează $(X^k, F(X^k))$
8: **end for**
9: **return** $(X^N, F(X^N))$

III.3.2.1 Cazul bi-dimensional

Pentru cazul bi-dimensional ($d = 2$) scopul algoritmului WRS este acela de a maximiza funcția $F : S_1 \times S_2 \to \mathbb{R}$, unde S_1 și S_2 sunt mulțimi finite. Restricția ca cele două mulțimi să fie finite derivă din faptul ca numărul de iterații pentru RS și WRS este finit, implicit numărul maxim de valori care pot fi testate este unul finit. Definesc ca *optim global* punctul $X^*(x_1^*, x_2^*)$, cu $x_1^* \in S_1$ și $x_2^* \in S_2$, astfel încât $F(X^*) \geq F(X), \forall X \in S_1 \times S_2$. $p_i, k_i, (i = 1, 2)$ sunt probabilitățile de schimbare și, respectiv, numărul minim de valori necesare pentru x_i, asa cum au fost definite anterior. $|S_i|$ este cardinalitatea lui $S_i, i = 1, 2$. Notez cu $p_{RS:n}$ și $p_{WRS:n}$ probabilitatea, în cazul RS și, respectiv, WRS, de a identifica optimul global după n încercări.

Următoarea teoremă stabilește că, în cazul bi-dimensional, putem alege k_2 astfel încât:

$$p_{WRS:n} \geq p_{RS:n} \tag{III.3.1}$$

Teoremă III.3.1. *Pentru orice funcție $F : S_1 \times S_2 \to \mathbb{R}$, S_1 și S_2 mulțimi finite, există k_2, astfel încât $p_{WRS:n} \geq p_{RS:n}$.*

III.3.2.2 Cazul multi-dimensional

Pentru cazul general al optimizării unei funcții $F : S_1 \times S_2 \ldots \times S_d \to \mathbb{R}$, unde $S_i, i = 1, \ldots, d$ sunt mulțimi finite, păstrând presupunerea că variabilele nu sunt corelate statistic, P_{RS} și P_{WRS} sunt definite astfel:

$$p_{RS} = \prod_{i=1}^{d} \frac{1}{|S_i|} \tag{III.3.2}$$

$$p_{WRS} = \frac{1}{|S_1|} \prod_{i=2}^{d} \left(p_i \frac{1}{|S_i|} + (1 - p_i) \frac{1}{|S_i| - m_i + 1} \right) \tag{III.3.3}$$

unde m_i este numărul de valori distincte deja generate pentru x_i.

Urmând același raționament ca în Secțiunea III.3.2.1, demonstrez următoarea teoremă:

Teoremă III.3.2. *Pentru orice funcție $F : S_1 \times S_2 \ldots \times S_d \to \mathbb{R}$, S_i mulțimi finite, există $k_i, i = 1, \ldots, d$, astfel încât: $p_{WRS:n} \geq p_{RS:n}$.*

Conform rezultatelor anterioare, pentru un set de valori $k_i, i = 1, \ldots, d$ ales corespunzător, la fiecare pas n, WRS are o probabilitate de a identifica optimul global mai mare decât RS. Astfel, după același număr de iterații, în medie, WRS identifică optimul global mai rapid decât RS, cu alte cuvinte, în medie, WRS converge mai rapid decât RS.

Mai mult decât atât, în cazul WRS, numărul de valori generate pentru $x_i, i = 1, \ldots, d$, urmează o distribuție binară cu probabilitatea p_i. După n pași, valoarea estimată pentru x_i, este așadar np_i. m_i are, deci, în medie, o limită superioară dată de np_i. Numărul de valori distincte generate depinde de cardinalitatea lui S_i și de distribuția de probabilitate folosită pentru generare.

De exemplu, în cazul distribuției uniforme, valoarea estimată pentru m_i este:

$$E[m_i] = \sum_{1}^{|S_i|} \left(1 - \left(\frac{|Si| - 1}{|Si|} \right)^{np_i} \right) \tag{III.3.4}$$

și $m_i > 1$ când $np_i > 1$. De aceea, pentru un număr de iterații n, cu $n \geq 1/p_i$, (III.3.1) este adevărată. Alegând k_i astfel încât $k_i > 1/p_i$, (III.3.1) este adevărată pentru orice valoare a lui n. Se poate observa, de asemenea, că diferența între $p_{WRS:n}$ și $p_{RS:n}$ crește când n crește.

III.3.2.3 Alegerea lui p_i și k_i

Am ales să ordonez variabilele funcției în ordinea importanței lor (pondere) și să asignez probabilitățile de schimbare p_i conform cu această ordine: cu cât ponderea unui parametru este mai mică, cu atât probabilitatea lui de schimbare este mai mică. Asftel, cel mai important parametru este acela care se va genera întotdeauna ($p_1 = 1$). Pentru a calcula ponderea fiecărui parametru, am rulat RS pentru un număr prestabilit de pași, $N_0 < N$ și am aplicat fANOVA [41] pentru valorile obținute. Am obținut astfel o estimare pentru importanța fiecărui parametru (hiperparametru, în cazul optimizării de hiperparametri). Dacă w_i este ponderea parametrului de pe poziția i și w_1 ponderea celui mai important parametru, atunci am ales $p_i = w_i/w_1, i = 1, \ldots, d$.

Pentru simplificare, am ales $k_i = N_0$ pentru toți parametrii dar aceste valori pot fi ajustate dependent de funcția obiectiv.

III.3.3 Un exemplu: Optimizarea funcției Griewank

Pentru a ilustra conceptul care stă la baza WRS, consider o funcție simplă, a cărei formă analitică este cunoscută. Faptul că valoarea funcției poate fi calculată rapid, permite testarea algoritmului pentru un număr foarte mare de execuții independente. Odată obținute rezultatele acestor execuții independente, ele pot constitui baza unui test t-student nepereche (unpaired t-test), test care permite eliminarea factorului aleator în compararea performanței obținute.

Funcția Grievank [34] este des folosită pentru a testa convergența algoritmilor de optimizare. Acestă funcție are următoarea formă analitică:

$$G_d = 1 + \frac{1}{4000}\sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d} \cos\frac{x_i}{\sqrt{i}} \tag{III.3.5}$$

Funcția este dificil de optimizat din cauza numărului foarte mare de minime locale. Am folosit pentru a testa WRS o versiune ușor modificată a funcției G_6, cu următoarea formă analitică:

$$G_6^* = 1 + \frac{i-1}{4000}\sum_{i=1}^{6} x_i^2 - \prod_{i=1}^{6} \cos\frac{x_i}{\sqrt{i}} \tag{III.3.6}$$

și am maximizat $-G_6^*$.

Am folosit $S = [-600, 600]$ pentru toti cei șase parametri și am rulat WRS pentru 1000 de iterații, cu o fază inițială de tip RS pentru primele $1000/e = 368$ iterații [26]. După terminarea primei faze am rulat fANOVA și am obținut ponderile parametrilor. Am comparat rezultatele obținute cu RS, folosind același spațiu de căutare și 1000 de încercări. Am rulat ambii algoritmi (RS și WRS) de 10.000 de ori. Tabela III.3.1 prezintă cel mai bun rezultat obținut de RS și respectiv WRS precum și media și abaterea standard pentru cele 10.000 de execuții.

Tabela III.3.1: WRS vs. RS - rezultatele pentru maximizarea $-G_6^*$ - valori pentru 10.000 de experimente cu 1000 de iterații fiecare

Algoritmul	Cel mai bun rezultat obținut	Media rezultatelor obținute	Abaterea standard pentru rezultatele obținute
RS	-1.50	-33.10	14.06
WRS	**-1.28**	**-14.58**	**10.63**

Rezultatele obținute de WRS sunt net superioare celor obținute cu RS. Eroarea standard pentru testul t-Student este, pentru rezultatele obținute 0.176 iar $df = 19998$, ceea ce duce la un nivel de semnificație P-value ≤ 0.001.

III.3.4 Optimizarea hiperparametrilor unei rețele de tip CNN

Secțiunea curentă prezintă rezultatele obținute de WRS pentru optimizarea arhitecturii unei rețele de tip CNN. Arhitecturile actuale identificate sunt complexe, au un număr impresionant de hiperparametri și un timp de antrenare foarte mare. Seturile de date folosite pentru antrenarea CNN sunt de asemenea de dimensiuni mari, ceea ce influențează negativ și mai mult timpul de antrenare. Îmbunătățirea performanței algoritmilor de optimizare de hiperparametri este, așadar, un subiect critic în contextul dat.

În cazul aplicării WRS pentru optimizarea de hiperparametri pentru o rețea CNN am considerat următorii hiperparametri:

- C - Numărul de straturi convoluționale: număr întreg în mulțimea $\{3, 4, 5, 6\}$;
- N - Numărul de straturi dense: număr întreg, în mulțimea $\{1, 2, 3, 4\}$;
- C-1,…,C-6 Numărul de convoluții pentru fiecare strat convoluțional: număr întreg în intervalul $[100, 1024]$;

Tabela III.3.2: Cea mai bună arhitectură CNN identificată pentru setul de date CIFAR-10

Alg.	C	D	C-1	C-2	C-3	C-4	C-5	C-6	N-1	N-2	N-3	N-4
WRS	6	1	736	508	664	916	186	352	1229	-	-	-
RS	5	1	876	114	892	696	617	-	1828	-	-	-
NM	5	3	564	564	564	560	563	-	1529	1542	1542	-
PSO	5	1	479	792	584	411	593	-	1379	-	-	-
SS	5	2	402	933	750	997	777	-	1545	1268	-	-

- D-1,…,D-4 Numărul de neuroni pentru fiecare strat dens: număr întreg în intervalul $[1024, 2048]$.

Am generat fiecare hiperparametru conform unei distribuții uniforme și am folosit acuratețea clasificării pentru a evalua performanța modelelor obținute.

Am folosit Keras [20] pentru a antrena și testa modelele CNN obținute. Am antrenat fiecare model pentru zece epoci pe setul de date CIFAR-10 [47]. Am rulat experimentele pe un cluster IBM S822LC cu noduri de tip IBM POWER8, NVLink și NVidia Tesla P100 GPUs[1]. Setul de date CIFAR-10 constă în 60.000 de imagini color cu dimensiunea de 32×32 pixeli, grupate egal în 10 clase (6.000 de imagini per clasă). Datele sunt împărțite în 50.000 de înregistrări pentru antrenare și 10.000 pentru testare. Am antrenat modelele pe 90% din datele de antrenare și am folosit restul de 10% ca date de validare. Am raportat rezultatele pe datele de test. Nu am folosit tehnici de augmentare a datelor.

Am comparat rezultatele obținute de WRS cu cele opbținute de RS, Nelder-Mead (NM), Particle Swarm (PSO) [44] și optimizarea de tip Secvențe Sobol(SS) [78] implementate în Optunity [62].

După prima etapă a algoritmului care constă în execuția a $300/e \approx 110$ iterații de tip RS, am rulat fANOVA pentru a obține ponderea fiecărui parametru. fANOVA consideră cei mai importanți hiperparametri (cei cu cea mai mare pondere în variația acurateței rețelei) ca fiind (în ordine descrescătoare a importanței): numărul de neuroni în primul strat dens (D-1), numărul de straturi dense (D) și numărul de straturi convoluționale (C). Ponderea majorității celorlalți hiperparametri este cu un ordin de magnitudine mai mică. În acest context, cea de-a două fază a WRS favorizează evident schimbarea celor mai importanți trei parametri în ordinea descrescătoare a ponderii lor.

WRS obține performanțe superioare comparativ cu toate celelalte metode (detalii

[1] http://www.cwu.edu/faculty/turing-cwu-supercomputer

în tabela III.3.3.

Tabela III.3.3: Acuratețea obținută de WRS, RS, NM, PSO și SS pentru optimizarea hiperparametrilor unei rețele CNN pe setul de date CIFAR-10

Algoritmul	Cel mai bun rezultat	Media rezultatelor obținute	Abaterea standard a rezultatelor obținute
WRS	**0.85**	**0.79**	0.09
RS	0.81	0.75	0.04
NM	0.81	0.77	0.03
PSO	0.83	0.78	0.03
SS	0.82	0.75	0.05

Tabela III.3.2 prezintă arhitectura optimă identificată de fiecare algoritm. Se poate observa că în cazul WRS și RS, arhitecturile obținute au un singur strat dens și un număr mare de straturi convoluționale (cinci pentru RS și șase pentru WRS)

III.3.5 Concluzii

Am introdus o variantă îmbunătățită a metodei de tip RS și anume algoritmul WRS. Folosind același buget computațional (număr de încercări), WRS converge mai rapid decât RS. Algoritmul prezentat obține rezultate superioare atât pentru optimizarea unei funcții matematice cunoscute cât și în cazul optimizării hiperparametrilor pentru o rețea de tip CNN. Transferul de informație necesar între iterațiile algoritmului este redus, așa cum se poate observa din descrierea Algoritmului 3. Datorită acestui fapt WRS poate fi ușor implementat în paralel. Deoarece nu am impus restricții funcției obiectiv, algoritmul WRS poate fi generalizat și în cazul altor probleme de optimizare definite pe un domeniu discret. O posibilă direcție de cercetare viitoare constă în aplicarea WRS pentru alte clase de probleme de optimizare, în special în contextul optimizării de hiperparametri în învățarea automată precum și compararea, în acest caz, cu rezultatele obținute de algoritmi din zona optimizării Bayesiene.

Partea IV

Concluzii finale, contribuții originale. Diseminarea rezultatelor. Direcții viitoare de cercetare

Capitolul IV.1
Concluzii finale, contribuții originale

Capitolul curent conține concluziile finale ale tezei cu accent pe contribuțiile originale prezentate. Prezenta teză este structurată pe două direcții majore de cercetare:

- Atribuirea automată a erorilor în proiectele mari de tip software open source - Partea a II-a
- Optimizarea hiperparametrilor în învățarea automată -Partea a III-a

În contextul atribuirii automate a erorilor în proiectele mari de tip software open source am introdus două sisteme de recomandare bazate pe SVM (Capitolul II.2) și respectiv tehnici de învățare profundă - CNN și LSTM (Capitolul II.3).

Sistemul de recomandare prezentat în Capitolul II.2 este **prima implementare paralelă pe o platformă de tip cloud** (Apache Spark găzduit pe o arhitectură de tip Google Cloud DataProc) pentru un astfel de sistem. Aceasta vine în contextul în care cantitatea de date disponibile este din ce în ce mai mare, ceea ce pune o presiune ridicată pe implementările standard de tip desktop. Raportat la rezultatele anterioare: Anvik *et al.* [5], [6] și [7], precizia și senzitivitatea obținute pe trei seturi de date ale unor proiecte reale sunt similare, aceasta în contextul unui sistem cu o scalabilitate ridicată, capabil să gestioneze volume de date din ce în ce mai mari. **Un alt aspect unic al implementării propuse este secvența de pași folosiți pentru preprocesarea datelor.** Mai specific, am folosit Stanford parts-of-speech (POS) pentru a identifica exclusiv substantivele din setul de date de intrare. Am folosit Term Frequency/Inverse Document Frequency (TF/IDF) pentru a transforma datele textuale în date numerice și am introdus informațiile legate de produs (`product_id`) și componentă (`component_id`) sub formă de valori ponderate.

Sistemul de recomandare descris în Capitolul II.3 este de asemenea **primul bazat pe tehnici de învățare profundă aplicat în domeniul atribuirii automate a erorilor**. Performanța în ceea ce privește calitatea predicției făcute este echivalentă cu cele mai bune rezultate obținute în prealabil în acest domeniu. Implementarea este, și în acest caz una paralelă și scalabilă, ceea ce răspunde eficient unui volum de date în continuă creștere. Un alt aspect original al implementării propuse constă în modul

în care datele textuale sunt folosite ca date de intrare pentru rețeaua de tip CNN. Pentru a induce datelor **un nivel mai ridicat de spațialitate** între cuvintele dintr-un paragraf, în cazul CNN, **am împărțit reprezentarea textuală a rapoartelor în mai multe secvențe egale și am calculat reprezentarea lor prin PV-DBOW individual**.

În zona optimizării hiperparametrilor pentru algoritmii de tip învățare automată, am propus două direcții de îmbunătățire a algoritmului standard de tip Căutare Aleatoare (RS). Prima (Capitolul III.2) are ca scop reducerea numărului de încercări necesar fără a influența negativ performanța sistemului, cea de-a doua (Capitolul III.3) propune îmbunătățirea rezultatului obținut folosind același număr de încercări.

În Capitolul III.2 este prezentat un **criteriu dinamic de oprire automată** pentru RS. Implementarea criteriului propus duce la o **reducere semnificativă a numărului de încercări** efectuate comparativ cu varianta clasică de RS fără a influența negativ rezultatul obținut. Mai mult, am propus **o variantă de implementare paralelă** a criteriului introdus, am demonstrat că această implementare este scalabilă și că **probabilitatea de a identifica rezultatul optim după un număr de încercări semnificativ redus, crește odată cu creșterea nivelului de paralelism**. Am testat algoritmul propus în contextul optimizării hiperparametrilor unui clasificator de tip SVM, pe șase din cele mai utilizate seturi de date disponibile public. Am comparat acuratețea raportată de modelele obținute atât cu RS cât și cu alți algoritmi cunoscuți. Conform testelor efectuate, criteriul propus accelerează semnificativ RS și obține rezultate comparabile cu celelalte tehnici de optimizare testate.

Capitolul III.3 prezintă de asemenea **o variantă îmbunătățită de RS**. De această dată, însă, focusul nu este pe reducerea numărului de încercări necesare ci pe **îmbunătățirea rezultatului obținut folosind același buget computațional**. Spre deosebire de algoritmul clasic RS, care, la fiecare iterație, generează valori noi pentru fiecare hiperparametru, algoritmul propus în Capitolul III.3 atribuie fiecărui hiperparametru o probabilitate de schimbare. La fiecare iterație, pentru fiecare hiperparametru, cu probabilitatea de schimbare aferentă, este generată o valoare nouă. Cu o probabilitate inversă, este folosită cea mai bună valoare obținută până la momentul respectiv. Intuiția din spatele acestei abordări este că o valoare care deja a dus la obținerea unui rezultat promițător, este un bun candidat pentru o încercare nouă, în combinație cu valori noi pentru ceilalți hiperparametri. Am denumit algoritmul propus Căutare Aleatoare Ponderată (Weighted Random Search - WRS). Am testat WRS atât pentru optimizarea unei variații a funcției Grievank cât și pentru optimizarea hiperparametrilor unei rețele neurale de tip CNN. În ambele cazuri, WRS obține rezultate semnificativ mai bune decât RS și alți algoritmi standard de optimizare, rezultate demonstrate și din punct de vedere teoretic.

Capitolul IV.2

Diseminarea rezultatelor. Direcții viitoare de cercetare

Rezultatele originale prezentate în Partea a II-a și Partea a III-a au fost prezentate la conferințe importante în domeniu și publicate în volume indexate Web of Science. Acest capitol prezintă aspecte legate de diseminarea rezultatelor și direcțiile viitoare de cercetare.

Rezultatele prezentate în Capitolul II.2 au fost prezentate în cadrul conferinței ”International Conference on Artificial Intelligence and Soft Computing” în 2017 și publicate în Springer, ”Lecture Notes in Computer Science” volumul 10246 cu titlul: ”Spark-based cluster implementation of a bug report assignment recommender system” [29]. Lucrarea este indexată Web of Science. Lucrarea descrie sistemul de recomandare propus cu accent pe caracterul paralel al implementării cât și pe secvența originală de tehnici folosite în preprocesarea datelor. Ca direcții viitoare de cercetare subliniez posibilitatea de a folosi tehnici similare de preprocesare a datelor de intrare pentru alte clase de probleme unde informația disponibilă este atât de natură textuală cât și categorială precum și implementarea de clasificatoare scalabile, folosind Apache Spark pentru alte domenii unde volumul datelor în continuă creștere devine problematic.

Rezultatele prezentate în Capitolul II.3 au fost prezentate în cadrul conferinței ”International Conference on Artificial Neural Networks” în 2017 și publicate în Springer, ”Lecture Notes in Computer Science” volumul 10614 cu titlul: ”Parallel Implementation of a Bug Report Assignment Recommender Using Deep Learning” [28]. Lucrarea este indexată Web of Science. Lucrarea este, din cunoștințele mele, prima implementare a unui sistem de recomandare în domeniul atribuirii automate a erorilor folosind tehnici specifice învățării profunde. Lucrarea deschide noi direcții de cercetare în zona folosirii altor arhitecturi de tip învățare profundă pentru rezolvarea acestui tip de probleme precum și în posibilitatea folosirii de GPU-uri pentru antrenarea modelului propus în scopul obținerii unei viteze de antrenare net superioare modelelor existente. Acest gen de modele cu timp de antrenare foarte redus dar cu performanțe ridicate în ceea ce privește rezultatele clasificării pot constitui baza unor sisteme de recomandare aproape în timp real. O

altă direcție posibilă de cercetare este dată de metoda originală prin care datele de intrare de tip textual sunt introduse într-o rețea de tip CNN, în mod stratificat (fiecare paragraf este împărțit în n secvențe de text de dimensiuni egale și fiecare secvență este convertită la reprezentarea ei numerică obținându-se astfel o reprezentare numerică bidimensională a textului de intrare). Acest gen de reprezentare conferă spațialitate reprezentării numerice a datelor textuale, spațialitate necesară în contextul rețelelor de tip CNN și poate fi adaptată și altor rețele CNN care folosesc ca date de intrare informații textuale.

Rezultatele prezentate în Capitolul III.2 au fost prezentate în cadrul conferinței "IFIP International Conference on Artificial Intelligence Applications and Innovations" în 2018 și publicate în Springer, "IFIP Advances in Information and Communication Technology" volumul 519 cu titlul: "A Dynamic Early Stopping Criterion for Random Search in SVM Hyperparameter Optimization" [26]. Criteriul de oprire dinamic introdus reduce semnificativ numărul de iterații necesare algoritmului RS, în contextul optimizării hiperparametrilor unui clasificator de tip SVM. Ca direcție de cercetare viitoare, cea mai importantă este aplicarea acestui criteriu în contextul optimizării de hiperparametri pentru alți algoritmi din învățarea automată, în special în zona învățării profunde, unde numărul mare de hiperparametri și timpul mare de antrenare pun probleme deosebite. Un astfel de criteriu ar putea permite, așa cum am demonstrat în cazul SVM, obținerea unor rezultate comparabile cu tehnicile cunoscute dar cu o reducere semnificativă a bugetului computațional necesar. Algoritmul propus este, de asemenea, suficient de flexibil pentru a putea fi adaptat oricărei probleme de optimizare definite pe un domeniu discret.

Rezultatele prezentate în Capitolul III.3 sunt publicate în International Journal of Computers Communications & Control (IJCCC), Nr 14(2), Aprilie 2019 - Factor de impact în JCR2017 (Clarivate Analytics, SCI Expanded, ISI Web of Science): IF=1.29, cu titlul "Weighted Random Search for Hyperparameter Optimization" [27]. Algoritmul propus, WRS, obține rezultate net superioare celorlalte metode testate atât în contextul optimizării unei variații a funcției Grievank cât și în cazul optimizării de hiperparametri pentru o rețea de tip CNN. Ca direcții viitoare de cercetare consider ca algoritmul este un bun candidat pentru a fi testat în cazul optimizării unui număr mult mai mare de hiperparametri cât și pentru alte arhitecturi de tip învățare profundă. Identificarea unei relații între valorile acestor probabilități de schimbare și funcția obiectiv, alta decât cea propusă - în funcție de importanța parametrilor - constituie o altă direcție posibilă de cercetare viitoare. Și în acest caz, algoritmul propus este, suficient de flexibil pentru a putea fi adaptat oricărei probleme de optimizare definite pe un domeniu discret.

Bibliografie

[1] Ahsan, S. N., Ferzund, J., and Wotawa, F. Automatic software bug triage system (bts) based on latent semantic indexing and Support Vector Machine. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on* (2009), 216–221.

[2] Albelwi, S., and Mahmood, A. A framework for designing the architectures of deep convolutional neural networks. *Entropy 19*, 6 (2017).

[3] Alpaydin, E. *Introduction to Machine Learning*, 2nd ed. The MIT Press, (2010).

[4] Alsheikh, M. A., Niyato, D., Lin, S., Tan, H., and Han, Z. Mobile big data analytics using deep learning and Apache Spark. *CoRR abs/1602.07031* (2016).

[5] Anvik, J. Automating bug report assignment. In *Proceedings of the 28th International Conference on Software Engineering* (New York, NY, USA, 2006), ICSE '06, ACM, 937–940.

[6] Anvik, J., Hiew, L., and Murphy, G. C. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering* (New York, NY, USA, 2006), ICSE '06, ACM, 361–370.

[7] Anvik, J., and Murphy, G. C. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Trans. Softw. Eng. Methodol. 20*, 3 (2011), 10:1–10:35.

[8] Banitaan, S., and Alenezi, M. Tram: An approach for assigning bug reports using their metadata. In *2013 Third International Conference on Communications and Information Technology* (2013), 215–219.

[9] Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *NIPS*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira, and K. Q. Weinberger, Eds. (2011), 2546–2554.

[10] Bergstra, J., and Bengio, Y. Random search for hyper-parameter optimization. *J. Mach. Learn. Res. 13* (2012), 281–305.

[11] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, (2006).

[12] Blei, D. M., Ng, A. Y., Jordan, M. I., and Lafferty, J. Latent Dirichlet Allocation. *Journal of Machine Learning Research 3* (2003).

[13] Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., and Zieba, K. End to end learning for self-driving cars. *CoRR abs/1604.07316* (2016).

[14] Buczak, A. L., and Guven, E. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys & Tutorials 18* (2016), 1153–1176.

[15] Cavalcanti, Y. C., da Mota Silveira Neto, P. A., Machado, I. d. C., Vale, T. F., de Almeida, E. S., and Meira, S. R. d. L. Challenges and opportunities for software change request repositories: a systematic mapping study. *Journal of Software: Evolution and Process 26*, 7 (2014), 620–653.

[16] Chan, P. K., Fan, W., Prodromidis, A. L., and Stolfo, S. J. Distributed data mining in credit card fraud detection. *IEEE Intelligent Systems 14* (1999), 67–74.

[17] Chang, C.-C., and Lin, C.-J. LIBSVM: A library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology 2* (2011), 27:1–27:27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[18] Chapelle, O., Vapnik, V., Bousquet, O., and Mukherjee, S. Choosing multiple parameters for Support Vector Machines. *Machine Learning 46*, 1 (2002), 131–159.

[19] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, (2014), 1724–1734.

[20] Chollet, F., et al. Keras. `https://keras.io`, (2015).

[21] Cortes, C., and Vapnik, V. Support-vector networks. *Mach. Learn. 20*, 3 (1995), 273–297.

[22] Cubranic, D., and Murphy, G. C. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), Banff, Alberta, Canada, June 20-24, 2004* (2004), 92–97.

[23] Demsar, J. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research 7* (2006), 1–30.

[24] DOMHAN, T., SPRINGENBERG, J. T., AND HUTTER, F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, AAAI Press, (2015), 3460–3468.

[25] FAN, R.-E., CHANG, K.-W., HSIEH, C.-J., WANG, X.-R., AND LIN, C.-J. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research 9* (2008), 1871–1874.

[26] **FLOREA**, A. C., AND ANDONIE, R. A Dynamic Early Stopping Criterion for Random Search in SVM Hyperparameter Optimization. In *14th IFIP International Conference on Artificial Intelligence Applications and Innovations (AIAI)* (Rhodes, Greece, May 2018), L. Iliadis, I. Maglogiannis, and V. Plagianakos, Eds., vol. AICT-519 of *Artificial Intelligence Applications and Innovations*, Springer International Publishing, (2018), 168–180. Part 3: Support Vector Machines.

[27] **FLOREA**, A. C., AND ANDONIE, R. Weighted random search for hyperparameter optimization. *International Journal of Computers Communications & Control 14* (2019), 154–169.

[28] **FLOREA**, A. C., ANVIK, J., AND ANDONIE, R. Parallel implementation of a bug report assignment recommender using deep learning. In *Artificial Neural Networks and Machine Learning – ICANN 2017* (Cham, 2017), A. Lintas, S. Rovetta, P. F. Verschure, and A. E. Villa, Eds., Springer International Publishing, (2017), 64–71.

[29] **FLOREA**, A. C., ANVIK, J., AND ANDONIE, R. Spark-based cluster implementation of a bug report assignment recommender system. In *Artificial Intelligence and Soft Computing* (2017), vol. 10246 of *LNAI*, Springer International Publishing, (2017), 31–42.

[30] FRIEDMAN, M. A comparison of alternative tests of significance for the problem of m rankings. *Ann. Math. Statist. 11*, 1 (1940), 86–92.

[31] GERS, F. A., AND SCHMIDHUBER, J. Recurrent nets that time and count. Tech. rep., (2000).

[32] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[33] GRAVES, A. Supervised sequence labelling with Recurrent Neural Networks. Studies in Computational intelligence. Springer, Heidelberg, New York, (2012).

[34] GRIEWANK, A. Generalized decent for global optimization. *Journal of Optimization Theory and Applications 34* (1981), 11–39.

[35] Hansen, N., Müller, S. D., and Koumoutsakos, P. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evol. Comput. 11*, 1 (2003), 1–18.

[36] Harris, D., and Harris, S. *Digital Design and Computer Architecture, Second Edition*, 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, (2012).

[37] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), 770–778.

[38] Hinton, G. E. A practical guide to training Restricted Boltzmann Machines. In *Neural Networks: Tricks of the Trade (2nd ed.)*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds., vol. 7700 of *Lecture Notes in Computer Science*. Springer, (2012), 599–619.

[39] Hochreiter, S., and Schmidhuber, J. Long Short-Term Memory. *Neural Comput. 9*, 8 (1997), 1735–1780.

[40] Holm, S. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics 6* (1979), 65–70.

[41] Hutter, F., Hoos, H., and Leyton-Brown, K. An efficient approach for assessing hyperparameter importance. In *Proceedings of International Conference on Machine Learning 2014 (ICML 2014)* (2014), 754–762.

[42] Hutter, F., Hoos, H. H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization* (Berlin, Heidelberg, 2011), LION'05, Springer-Verlag, (2011), 507–523.

[43] Iman, R., and Davenport, J. Approximations of the critical region of the Friedman statistic. *Communications in Statistics-Theory and Methods 9* (1980), 571–595.

[44] Kennedy, J., and Eberhart, R. C. Particle Swarm Optimization. In *Proceedings of the IEEE International Conference on Neural Networks* (1995), 1942–1948.

[45] Kirkpatrick, S. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics 34*, 5 (1984), 975–986.

[46] Kononenko, I. Machine learning for medical diagnosis: History, state of the art and perspective. *Artif. Intell. Med. 23*, 1 (2001), 89–109.

[47] Krizhevsky, A., Nair, V., and Hinton, G. Cifar-10 (canadian institute for advanced research).

[48] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., (2012), 1097–1105.

[49] Le, Q. V., and Mikolov, T. Distributed representations of sentences and documents. *CoRR abs/1405.4053* (2014).

[50] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE* (1998), 2278–2324.

[51] LeCun, Y., Bottou, L., Orr, G., and Müller, K. *Efficient Backprop*, vol. 7700 LECTURE NO of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer, 2012, 9–48.

[52] Lemley, J., Jagodzinski, F., and Andonie, R. Big holes in big data: A Monte Carlo algorithm for detecting large hyper-rectangles in high dimensional data. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)* (2016), vol. 1, 563–571.

[53] Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR abs/1603.06560* (2016).

[54] Metsis, V., and et al. Spam filtering with Naive Bayes – which Naive Bayes? In *The Third Conference on Email and Anti-Spam (CEAS* (2006).

[55] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings* (2013).

[56] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., (2013), 3111–3119.

[57] Moritz, P., Nishihara, R., Stoica, I., and Jordan, M. I. SparkNet: Training Deep Networks in Spark. *ArXiv e-prints* (2015).

[58] Nasim, S., Razzaq, S., and Ferzund, J. Automated change request triage using alpha frequency matrix. In *Frontiers of Information Technology (FIT), 2011* (2011), 298–302.

[59] Nelder, J. A., and Mead, R. A Simplex Method for Function Minimization. *Computer Journal 7* (1965), 308–313.

[60] Nemenyi, P. *Distribution-free Multiple Comparisons*. Thesis Princeton University, (1963).

[61] Nguyen, T. T., Nguyen, A. T., and Nguyen, T. N. Topic-based, time-aware bug assignment. *SIGSOFT Softw. Eng. Notes 39*, 1 (2014), 1–4.

[62] [online]. Optunity. `http://optunity.readthedocs.io/en/latest/`. Accesat: 2017-09-01.

[63] [online] Apache Spark. Apache Spark FAQ, 2019. Accesat: 2019-03-16.

[64] [online] Google. The Go programming language. `https://golang.org/`. Accesat: 2017-09-01.

[65] [online] Google. Google HyperTune. Accesat: 2018-10-28.

[66] [online] Wikipedia. Deep learning wikipedia page. Accesat: 2019-03-03.

[67] Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web, (1998).

[68] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[69] Quinn, M. J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, (2003).

[70] Rajaraman, A., and Ullman, J. D. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, (2011).

[71] Ramos, J. Using TF-IDF to determine word relevance in document queries, (1999).

[72] Reis, C. R., de Mattos Fortes, R. P., Pontin, R., and Fortes, M. An overview of the software engineering process and tools in the Mozilla project, (2002).

[73] Rong, X. Word2Vec parameter learning explained. *arXiv preprint arXiv:1411.2738* (2014).

[74] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. MIT Press, Cambridge, MA, USA, 1986, ch. Learning Internal Representations by Error Propagation, (1986), 318–362.

[75] Shokripour, R., Anvik, J., Kasirun, Z. M., and Zamani, S. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (Piscataway, NJ, USA) MSR '13, IEEE Press, (2013), 2–11.

[76] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2014).

[77] Smusz, S., Czarnecki, W. M., Warszycki, D., and Bojarski, A. J. Exploiting uncertainty measures in compounds activity prediction using Support Vector Machines. *Bioorganic & medicinal chemistry letters 25*, 1 (2015), 100–105.

[78] Sobol, I. Uniformly distributed sequences with an additional uniform property. *USSR Computational Mathematics and Mathematical Physics 16*, 5 (1976), 236 – 242.

[79] Sokolova, M., and Lapalme, G. A systematic analysis of performance measures for classification tasks. *Inf. Process. Manage. 45*, 4 (2009), 427–437.

[80] Srinivasan, A., Mascagni, M., and Ceperley, D. Testing parallel random number generators. *Parallel Comput. 29*, 1 (2003), 69–94.

[81] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)* (2015).

[82] Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA), KDD '13, ACM, (2013) ,847–855.

[83] Toutanova, K., Klein, D., Manning, C. D., and Singer, Y. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1* (Stroudsburg, PA, USA, 2003), NAACL '03, Association for Computational Linguistics, (2003), 173–180.

[84] Vapnik, V., and Lerner, A. Pattern recognition using generalized portrait method. *Automation and Remote Control 24* (1963).

[85] Yang, Y., and Pedersen, J. O. A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning* (San Francisco, CA, USA), ICML '97, Morgan Kaufmann Publishers Inc., (1997), 412–420.

[86] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA), NSDI'12, USENIX Association, (2012), 2–2.

[87] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, (2010), 10–10.

[88] Zeiler, M. D., and Fergus, R. Visualizing and understanding convolutional networks. In *Computer Vision – ECCV 2014* (Cham, 2014), D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds., Springer International Publishing, (2014), 818–833.

[89] Zoph, B., and Le, Q. V. Neural architecture search with reinforcement learning. *CoRR abs/1611.01578* (2016).

[90] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. *CoRR abs/1707.07012* (2017).

Anexe

Anexa1. Scurt rezumat al tezei

Teza intitulată **Sisteme Inteligente de Decizie** este structurată în patru parți. Prima parte este o parte introductivă, unde sunt prezentate noțiuni legate de învățarea automată cu accent pe tehnicile folosite și anume Paragraph2Vec, SVM, CNN și LSTM. Sunt de asemenea prezentate tehnicile de generare paralelă a numerelor pseudo-aleatoare, tehnici folosite în capitolul II.2. Capitolul I.4 descrie câțiva din cei mai des folosiți algoritmi de optimizare de hiperparametri, algoritmi folosiți ca referință în Partea a III-a. Prima parte se încheie cu o scurtă descriere a bibliotecii software Apache Spark, folosită în Partea a II-a.

Partea a II-a constă în trei capitole și prezintă rezultatele originale obținute în domeniul atribuirii automate a erorilor în proiectele mari de tip software open source. Primul capitol este un capitol introductiv unde este descrisă în linii mari problema atribuirii automate de erori cât și lucrările relevante în acest domeniu. Capitolul II.2 prezintă un sistem inteligent de decizie (recomandare) pentru acest tip de problemă, bazat pe SVM. Sistemul de recomandare este prima implementare paralelă pe o arhitectură de tip cloud destinată rezolvării acestei probleme. Capitolul II.3 prezintă un sistem de decizie bazat pe tehnici specifice învățării profunde, mai exact CNN și LSTM. Implementarea propusă este, și în acest caz, una paralelă și scalabilă. Ambele sisteme de recomandare obțin rezultate similare celor raportate în literatura de specialitate pe seturi de date de test corespunzătoare unor proiecte reale (Eclipse, Mozilla și Netbeans) în contextul unei scalabilități ridicate, ceea ce permite gestionarea unui volum de date în continuă creștere.

Partea a III-a constă, de asemenea, în trei capitole și prezintă rezultatele originale obținute în domeniul optimizării hiperparametrilor pentru învățarea automată. Primul capitol din această parte este un capitol introductiv, unde este descrisă problema optimizării hiperparametrilor și sunt prezentate lucrările semnificative din acest domeniu. Capitolele III.2 și III.3 prezintă două abordări diferite pentru optimizarea algoritmului de tip RS. Prima abordare constă în introducerea unui criteriu dinamic de oprire automată, criteriu care permite obținerea de rezultate similare celor raportate de implementările consacrate (RS, Grid Search, Nelder-Mead, Particle Search) după un număr de încercări semnificativ redus.

Am prezentat și o variantă de implementare paralelă a algoritmului propus și am demonstrat că performanța algoritmului crește cu numărul de nuclee folosite. Cea de-a două direcție de cercetare în zona optimizării de hiperparametri constă în introducerea unei variante de RS care, în cadrul aceluiași buget computațional, obține rezultate semnificativ mai bune în comparație cu algoritmii standard (RS, Nelder-Mead, Particle-Search, Secvențe Sobol). Am testat algoritmul propus atât în cazul optimizării unei variații a funcției Grievank cât și în cazul optimizării hiperparametrilor unei rețele neurale de tip CNN. Am demonstrat, de asemenea, din punct de vedere teoretic că algoritmul propus converge mai repede decât RS la optimul funcției obiectiv.

Partea a IV-a prezintă concluziile finale ale tezei, cu accent pe contribuțiile originale. Este prezentat modul de diseminare a rezultatelor și sunt menționate câteva direcții posibile de cercetare plecând de la rezultatele originale descrise.

* *
*

The current PhD thesis entitled **Sisteme Inteligente de Decizie** (Intelligent Decision Systems) is structured in four parts. The first part in an introductory one. There are presented introductory aspects regarding Machine Learning, focusing on the utilized techniques: Paragraph2Vec, SVM, CNN and LSTM. There are also presented the parallel random number generation techniques utilized in Chapter II.2. Chapter I.4 describes some of the most commonly used hyperparameter optimization algorithms, algorithms used as reference in the III-rd part. The last chapter of the first part consists in a short description of the Apache Spark software library, library used in the II-nd part of the thesis.

The II-nd part consists in three chapters and presents the original results obtained in the field of automatic assignment of errors in large Open Source software projects. The first chapter is an introductory one and describes the automatic bug triage problem as well as the relevant work in this area. Chapter II.2 describes an intelligent decision (recommender) system for this type of problem, based on SVM. The recommender system is the first parallel implementation, on a cloud architecture, designed to solve this type of problem. Chapter II.3 present a bug triage decision system based on Deep Learning techniques, specifically CNN and LSTM. The proposed implementation is, in this case also, a parallel and scalable one. Both recommenders obtain on par results with the state of the art implementations on real life data sets from Eclipse, Mozilla and Netbeans and they achieve a high level of scalability, which allows the handling of an ever increasing volume of available data.

The III-rd part consists, also, in three chapters and presents the original results obtained in the field of hyperparameters optimization for Machine Learning. The first chapter of this part is an introductory one and describes the problem of hyperparameter optimization. The state of the art regarding this topic is also presented. Chapters III.2 and III.3 describe two distinct approaches on optimizing the RS algorithm. The first approach consists in defining a dynamic early stopping criterion that allows achieving on par results with the ones reported by standard techniques (RS, Grid Search, Nelder-Mead, Particle Search) after a significantly reduced number of trials. I've also presented a possible parallel implementation for the proposed algorithm and proved that the parallel algorithm's performance increases with the number of utilized cores. The second research direction consists in introducing a modified variant of RS which, within the same computational budget, obtains significantly better results compared to the standard techniques (RS, Nelder-Mead, Particle-Search, Sobol Sequences). I have tested the proposed algorithm both for the optimization of a variation of a well known mathematical function, namely the Grievank functions, as well as for the optimization of the hyperparameters for a CNN type neural network. I have also proved, from a theoretical perspective, that the proposed algorithm converges faster than RS to the target function optimum.

The IV-th part presents the final conclusions of the thesis, with an accent on the original contributions. The way the original results are disseminated is listed and some further research directions are also proposed.

Anexa 2. Curriculum Vitae

Informații personale	**Adrian-Cătălin FLOREA** E-mail: acflorea@unitbv.ro
Experiența profesională	**2018-prezent** – Inginer de sistem software, S.C. CrowdStrike S.R.L. Dezvoltare de aplicații în domeniul securității cibernetice, pe o arhitectură de tip micro-servicii în cloud, folosind: - GOLANG, Python, Scala, - Cassandra, Elasticsearch, Kafka etc. **2014-2018** – Arhitect software / Manager de proiect informatic, S.C. Pentalog Romania, S.R.L Concepția arhitecturii pentru proiectele software dezvoltate în special în zona Big Data folosind: - Java, Scala, Python - Titan, Cassandra, Elasticsearch, Kafka etc. **2008-2014** – Programator, S.C. Pentalog Romania, S.R.L. Dezvoltarea de produse software diverse, folosind: - Java, Scala, Python, - Kafka, Hadoop, MySQL, Postgresql etc. **2001-2008** – Programator, S.C. Waters Romania S.R.L. Dezvoltarea de produse software folosind: - Java, EJB, Swing, - Oracle, IBM Websphere AS
Educație	**2014-prezent** – Doctorand, Domeniul Calculatoare și Tehnologia Informației, Facultatea de Inginerie Electrică și Știința Calculatoarelor, Universitatea *Transilvania*, Brasov **2001-2002** - Studii Aprofundate în specializarea Algoritmi și Produse Software Facultatea de Științe, Universitatea *Transilvania*, Brașov **1997-2001** – Licență în Informatică, Facultatea de Științe, Universitatea *Transilvania*, Brașov

Personal information **Adrian-Cătălin FLOREA**
E-mail: acflorea@unitbv.ro

Work experience

2018-prezent – Senior Software Engineer, S.C. CrowdStrike S.R.L.
Developing cloud base microservices,
in the cyberecurity area, using :
- GOLANG, Python, Scala,
- Cassandra, Elasticsearch, Kafka etc.

2014-2018 – Software Architect, Project Manager
S.C. Pentalog Romania, S.R.L
Conception for various projects architecture
especially in Big Data area using:
- Java, Scala, Python
- Titan, Cassandra, Elasticsearch, Kafka etc.

2008-2014 – Software Developer, S.C. Pentalog Romania, S.R.L.
Development of various software projects using:
- Java, Scala, Python,
- Kafka, Hadoop, MySQL, Postgresql etc.

2001-2008 – Software Developer, S.C. Waters Romania S.R.L.
Development of various software projects using:
- Java, EJB, Swing,
- Oracle, IBM Websphere AS

Education

2014-prezent – PhD student in Computers and Information Technology,
Faculty of Electrical Engineering and Computer Science,
Transilvania University of Brașov

2001-2002 - Advanced Studies in Algorithms and Software Products
Faculty of Sciences, Transilvania University of Brașov
1997-2001 – Bachelor degree in Information Technology,
Faculty of Sciences,
Transilvania University of Brașov