UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA TEHNOLOGIA INFORMAȚIEI

Lucrare de licență

Algoritmi de clasificare a traficului în rețelele de calculatoare

Absolvent

Lazăr William Patrick

Coordonator științific
Lect. Dr. Silviu – Laurențiu Vasile

București, iunie 2025

Rezumat

Această lucrare urmărește explorarea și compararea unor metode actuale de clasificare aplicate traficului de rețea. Sunt dezvoltate și testate trei modele diferite – Random Forest, XGBoost și o rețea neuronală artificială – utilizând un set de date, care conține multiple tipuri de trafic, inclusiv activități malițioase. Etapele principale includ selecția caracteristicilor, tratarea dezechilibrelor de clasă și ajustarea hiperparametrilor. Prin această abordare, sunt analizate performanțele modelelor într-un cadru practic, cu scopul de a identifica soluții fiabile pentru detecția intruziunilor în rețele.

Abstract

This paper explores and compares several classification techniques applied to network traffic analysis. Three distinct models – Random Forest, XGBoost, and an artificial neural network – are implemented and tested using a dataset that includes various types of traffic, both benign and malicious. The methodology involves feature selection, class imbalance handling, and hyperparameter tuning. The goal is to evaluate the practical performance of these models and to identify reliable solutions for intrusion detection in computer network.

Cuprins

Cuprins

1. Introducere

În ultimii ani, utilizarea Internetului la nivel global a crescut exponențial, cu peste 5,3 miliarde de utilizatori activi, reprezentând 66% din populația mondială [1]. Această creștere accelerată este însoțită de o creștere alarmantă a atacurilor cibernetice, afectând astfel infrastructurile critice, dar și securitatea informațiilor. Conform raportului *Verizon Data Breach Investigations Report 2023*, atacurile de tip DoS (*Denial of Service*) și DDoS (*Distributed Denial of Service*) sunt printre cele mai frecvente și au afectat peste 70% din organizațiile globale în 2022.

Pentru a combate eficient aceste amenințări, un rol esențial este purtat de ***Intrusion Detection Systems*** **(IDS)**, al căror scop este protejarea rețelelor prin identificarea traficului atacator și semnalarea automată a potențialelor amenințări către administratorii de rețea. Acestea se împart, în general, în două categorii: sisteme bazate pe semnături și sisteme bazate pe comportament. Cele bazate pe semnături identifică atacurile comparând traficul cu o bază de date de semnături cunoscute, însă au dificultăți în detectarea atacurilor *zero-day*. În schimb, sistemele bazate pe comportament, datorită utilizării tehnicilor de învățare automată (*Machine Learning*) pentru analiza tiparelor traficului și identificarea de anomalii, au capacitatea de a detecta atacurile noi, necunoscute, inclusiv în traficul criptat prin SSL/TLS (*Secure Socket Layer / Transport Layer Security*) [2, pg. 1], unde conținutul mesajelor este inaccesibil fără o cheie de decriptare.

1.1. Scopul și motivația lucrării

Scopul acestei lucrări este de a analiza, de a compara performanța și de a implementa diferiți algoritmi de clasificare a traficului din rețea, în contextul în care securitatea rețelelor devine o provocare din ce în ce mai mare. Motivația principală este de a găsi o soluție cât mai eficientă pentru a răspunde provocării detectării și clasificării atacurilor cibernetice într-o durată de timp rezonabilă, în același timp fără a emite un număr prea mare de alarme false.

În plus, această lucrare își propune să evalueze eficiența algoritmilor în ceea ce privește viteza de procesare, consumul de resurse și capacitatea de adaptare la scenariile de atac în timp real.

1.2. Contribuția proprie în realizarea lucrării

Lucrarea reflectă contribuția personală în toate etapele esențiale, de la pregătirea datelor până la evaluarea modelelor. Am analizat și curățat setul de date CICIDS2017, corectând valori invalide, coloane duplicate și probleme de codificare care afectau etichetele. De asemenea, am aplicat metode de echilibrare a claselor pentru a obține un set de date mai stabil și mai potrivit pentru antrenare.

Am realizat selecția caracteristicilor relevante pentru fiecare model, astfel încât să reduc complexitatea și să îmbunătățesc performanța generală. Modelele au fost implementate, antrenate și testate individual, iar rezultatele au fost comparate în funcție de mai multe metrice, urmărind atât acuratețea, cât și comportamentul în cazul claselor mai puțin reprezentate. Toate deciziile legate de preprocesare, alegerea algoritmilor și interpretarea rezultatelor au fost luate în urma documentării și testării proprii.

1.3. Structura lucrării

Lucrarea este structurată în patru capitole, dintre care două reprezintă părțile esențiale: capitolul teoretic și cel aplicativ. Capitolul 2 oferă o prezentare clară a algoritmilor analizați (Random Forest, XGBoost și Rețele Neuronale), alături de conceptele fundamentale din clasificare și învățare automată, precum și metricile folosite în evaluare. Capitolul 3 reunește partea practică a lucrării, incluzând prelucrarea datelor, selecția caracteristicilor, antrenarea și optimizarea modelelor, precum și analiza comparativă a rezultatelor obținute.

2. Fundamente teoretice

În acest capitol voi prezenta conceptele fundamentale teoretice legate de detecția anomaliilor din rețea și modelele de învățare automată utilizate pentru clasificarea traficului. Voi explora diferite tehnici de *Machine Learning*, dar și de *Deep Learning*.

2.1. Modele de Invatare Automata

Învățarea automată poate fi caracterizată drept „o știință (dar și o artă) a calculatoarelor, prin care acestea pot fi antrenate să învețe din datele primite, fără a fi programate specific pentru fiecare sarcină” [4, pg. 2]. Procesul constă în antrenarea unui model pe un set de date denumit **training set**, fiind apoi evaluat pe un set separat, denumit **test set**, extrăgând astfel automat modele și relații din datele furnizate. Aceste avantaje sunt evidențiate în situațiile în care metodele tradiționale sunt ineficiente, precum analiza seturilor mari și complexe de date, unde identificarea tiparelor este dificil de realizat.

2.1.1. Random Forest

Random Forest reprezintă una dintre cele mai utilizate metode [4, pg. 175] de învățare automată bazate pe ansambluri (*ensamble*), atât în clasificare, cât și în regresie, funcționând prin combinarea predicțiilor mai multor arbori de decizie (*decision trees*), depășind astfel limitările arborilor individuali, precum sensibilitatea la zgomot și la *overfitting*. Conform lucrării lui Breiman [5, abstract]: „fiecare arbore depinde de valorile unui vector aleator, eșantionat independent pentru fiecare arbore din pădure, ceea ce reduce corelarea dintre arbori și îmbunătățește generalizarea modelului”.

Procesul de antrenare presupune următorii pași:

- **Bootstrap Aggregation (Bagging):** presupune crearea mai multor subseturi din setul de date original, prin eșantionare cu înlocuire. În acest mod, fiecare arbore va fi antrenat pe un subset diferit, contribuind la reducerea variației și la creșterea stabilității [6, abstract].

- **Selecția aleatoare a caracteristicilor (Feature Bagging):** la fiecare nod al arborelui sunt selectate, în mod aleator, *m* caracteristici din totalul *p*. Valoarea optimă este $\sqrt{\boldsymbol{p}}$ pentru clasificare și $\frac{\boldsymbol{p}}{\boldsymbol{3}}$ pentru regresie, aceasta fiind o convenție utilizată în implementările moderne ale Random Forest. Acest proces reduce corelația dintre

arbori și îmbunătățește generalizarea modelului [7, pg. 592].

- **Agregarea predicțiilor:** fiecare arbore propune o anumită clasă, iar votul majoritar determină predicția finală. În cazul regresiei, rezultatul este determinat de media predicțiilor individuale [5, pg. 6].

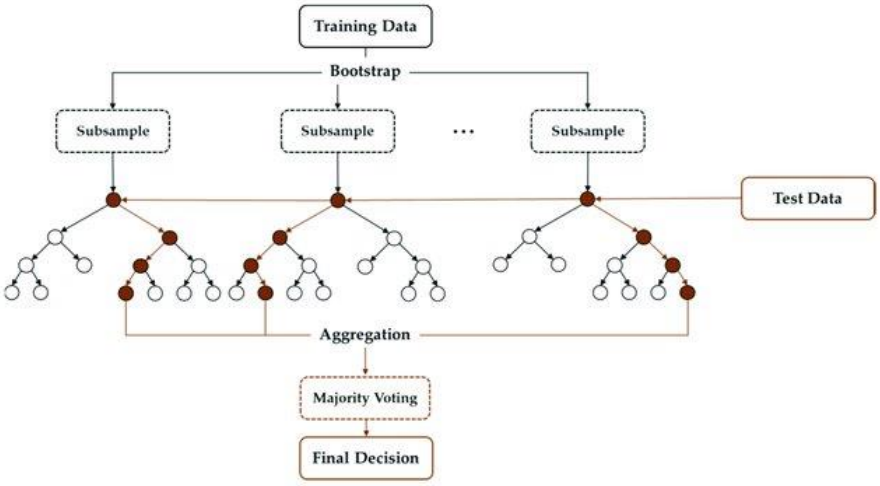


Figura 2.1. Ilustrarea funcționalității algoritmului Random Forest [29].

Criterii de împărțire

Pentru fiecare arbore de decizie, împărțirea nodurilor se realizează pe baza unor criterii de impuritate, cele mai frecvent utilizate fiind [7, pg. 309] indicele Gini și entropia. Indicele Gini reflectă impuritatea unui nod – o valoare mai mică indică o separare mai bună între clase, ceea ce corespunde unui nod mai „pur". Entropia, pe de altă parte, măsoară nivelul de incertitudine într-un nod; cu cât această valoare este mai mică, cu atât impuritatea nodului este mai redusă, indicând un criteriu de împărțire mai eficient. Formulele corespunzătoare celor două criterii sunt prezentate în Anexa G: formula F.1 pentru calculul Gini și formula F.2 pentru entropie.

Importanța Caracteristicilor

În Random Forest, importanța internă a caracteristicilor în timpul antrenării (*Feature Importance*) se calculează prin două metode:

- **Descresterea Medie a Impurității (*Mean Decrease in Impurity - MDI*)**: pentru fiecare caracteristică, măsoară contribuția la reducerea impurității în timpul antrenării arborilor [7, pg. 593].

- **Permutarea valorilor pe eșantioanele neutilizate (*Out-Of-Bag Permutation*)**: pentru fiecare caracteristică se modifică aleator valorile, în cadrul unui subset de date care nu a fost folosit la antrenare (setul OOB). Se compară apoi performanța modelului înainte și după permutare; dacă acuratețea scade semnificativ, caracteristica este considerată relevantă, iar o scădere minoră indică o contribuție redusă pentru predicție [7, pg. 593].

Performanța finală a modelului depinde de alegerea hiperparametrilor, care pot fi optimizați prin tehnicile de *GridSearch* sau *RandomSearch* [8]. Printre cei mai importanți hiperparametri se enumeră: **n_estimators**, care controlează numărul arborilor și influențează stabilitatea și varietatea modelului [7, pg. 197]; adâncimea maximă (**max_depth**), care limitează adâncimea arborilor și ajută la reducerea supraînvățării; numărul minim de eșantioane (**min_samples_split**), care stabilește câte instanțe sunt necesare pentru a împărți un nod; și **max_features**, care definește numărul maxim de caracteristici disponibile la fiecare împărțire.

Random Forest are o capacitate excelentă în gestionarea seturilor de date complexe. De asemenea, tolerează valorile lipsă din seturile de date, putând să le înlocuiască, și are un risc scăzut de supraînvățare, datorită modului său de funcționare.

2.1.2. eXtreme Gradient Boosting (XGBoost)

Extreme Gradient Boosting este o metodă bazată pe **arbori de decizie**, concepută pentru a îmbunătăți progresiv performanța unui model printr-o abordare iterativă. Este printre cele mai populare și eficiente tehnici de *Gradient Boosting*, remarcându-se prin capacitatea de scalare, viteza mare de execuție, reușind să fie de peste zece ori mai rapidă decât alte soluții pe o singură mașină și să gestioneze seturi de date foarte mari în medii distribuite sau cu resurse limitate [10, pg. 1, abstract].

Gradient Boosting construiește un model prin adăugarea succesivă de **arbori de decizie slabi** (***weak learners***), fiecare având rolul de a corecta erorile și de a îmbunătăți predicțiile arborilor anteriori. În XGBoost, este folosită o **funcție obiectiv regularizată**, care combină eroarea de predicție cu un termen de penalizare aplicat complexității modelului, prevenind astfel supraînvățarea. Forma generală a acestei funcții este prezentată în Anexa G, Formula F.3.

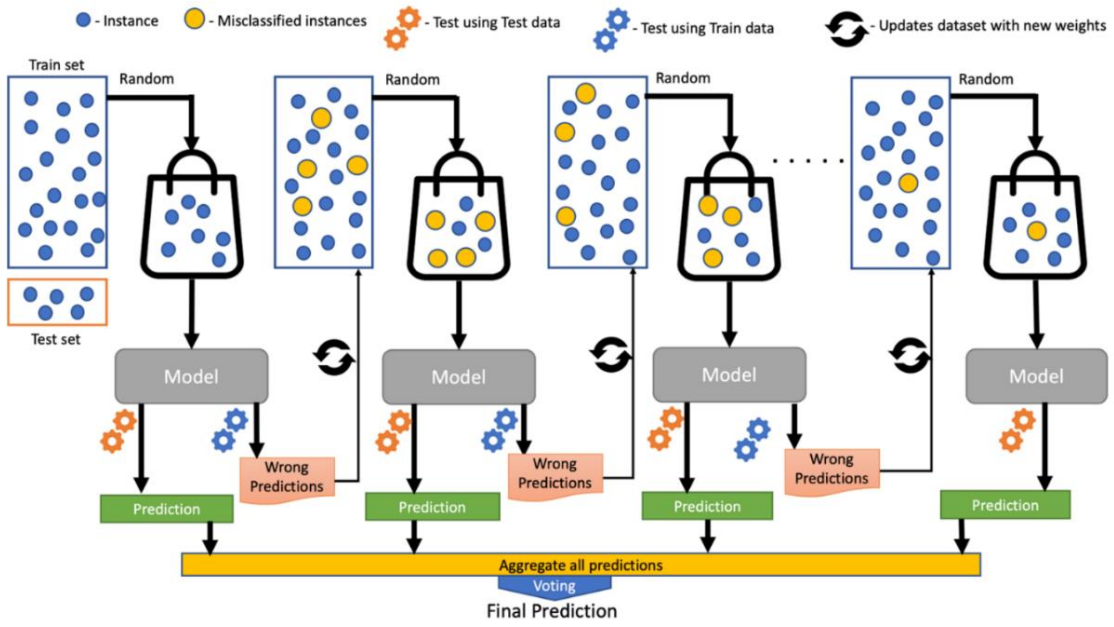


Figura 2.2. Ilustrarea funcționalității algoritmului Gradient Boosting: fiecare model corectează erorile modelului precedent, iar predicția finală este obținută prin agregarea tuturor predicțiilor [30].

Spre deosebire de alte implementări ale *Gradient Boosting*-ului, XGBoost utilizează o regularizare dublă, de tip L1 (*Lasso*) și L2 (*Ridge*), pentru a controla complexitatea arborilor. Formula completă a termenului de regularizare este prezentată în Anexa G, formula F.4, incluzând penalizările pentru numărul de noduri și pentru valorile ponderilor atribuite frunzelor.

Un avantaj important al XGBoost este reprezentat de **gestionarea automată a valorilor lipsă** fără a mai fi nevoie de preprocesare manuală a datelor. Algoritmul învață **direcția optimă de împărțire** în arbore pentru instanțele care au date lipsă, folosind o strategie denumită „***sparsity***

aware split finding" [10, sect. 3.4]. De asemenea, avem prezentă tehnica de **oprire anticipată** (***Early Stopping***), care oprește automat antrenarea modelului când performanța pe setul de validare nu se mai îmbunătățește după un anumit număr de iterații, reducând astfel timpul de execuție și prevenind *overfitting*-ul [4, pg. 141].

Performanța finală a modelului este corelată cu setarea optimă a hiperparametrilor, care pot fi ajustați automat prin tehnica de optimizare Optuna. Printre cei mai importanți se enumeră [11]: numărul de arbori antrenați (**n_estimators**), adâncimea maximă (**max_depth**), contribuția fiecărui arbore nou (**learning_rate**), **lambda** și **alpha** (factorii de regularizare L2 și L1 care controlează complexitatea și sensibilitatea), proporția de date folosite pentru fiecare arbore (**subsample**), valorile mici reduc *overfitting*-ul, dar pot scădea acuratețea – și proporția de caracteristici selectate aleator pentru fiecare arbore (**colsample_bytree**).

Nu în ultimul rând, modelul are capacitatea de a estima importanța caracteristicilor folosind **metoda câștigului** (***Gain method***) [10, pg. 3], care măsoară cât de mult contribuie fiecare caracteristică la reducerea erorii. O caracteristică este cu atât mai importantă cu cât determină o scădere mai mare a pierderii în momentul în care este utilizată într-un nod de decizie. Formula utilizată pentru calculul scorului de câștig este prezentată în Anexa G, formula F.5.

2.1.3. Deep Learning

Deep Learning (DL) reprezintă o revoluție în domeniul inteligenței artificiale, având capacitatea de a învăța relațiile complexe din date, fiind bazat pe **rețele neuronale artificiale**. Spre deosebire de metodele clasice de învățare automată, acestea au capacitatea de a extrage caracteristici abstracte și ierarhice din date, eliminând astfel necesitatea prelucrării manuale a caracteristicilor (*feature engineering*). *Deep Learning* utilizează straturi de transformare neliniare pentru a învăța reprezentările direct din datele brute [12, pg. 436].

Rețele Neuronale Artificiale

O rețea neuronală artificială, ANN (*Artificial Neural Network*), este alcătuită din unități interconectate, denumite neuroni artificiali. Acești neuroni sunt organizați în mai multe straturi

succesive: strat de intrare (*input*), unul sau mai multe straturi ascunse (*hidden*) și un strat de ieșire (*output*). Stratul de intrare preia datele brute, fiecare caracteristică fiind asociată unui neuron; straturile ascunse aplică transformări asupra datelor prin ponderi ajustabile și funcții de activare, iar stratul de ieșire generează rezultatul final, fie sub formă de clasă, fie sub formă de valoare continuă.

Pentru a procesa semnalele primite de la stratul anterior, fiecare neuron aplică o combinație liniară a intrărilor, exprimată ca $\boldsymbol{z = w^T x + b}$, unde **w** reprezintă ponderile, **x** intrările, iar **b** este termenul de *bias*. Rezultatul **z** este apoi trecut printr-o funcție de activare neliniară **a = f(z)**, cum ar fi *ReLU* sau *sigmoid*, care introduce complexitate și permite rețelei să modeleze relații nelineare.

Antrenarea unei rețele neuronale presupune două etape: *forward propagation*, în care datele sunt transmise prin rețea până la stratul de ieșire, și *backpropagation*, în care erorile sunt propagate înapoi și se ajustează ponderile pentru a reduce diferența dintre predicție și valoarea reală. Astfel, neuronii ascunși ajustează iterativ ponderile conexiunilor pentru a se minimiza diferența dintre predicție și eticheta din setul de antrenament, rezultatul real. În acest mod, rețeaua învață reprezentări utile ale datelor și își îmbunătățește capacitatea de generalizare [13, abstract].

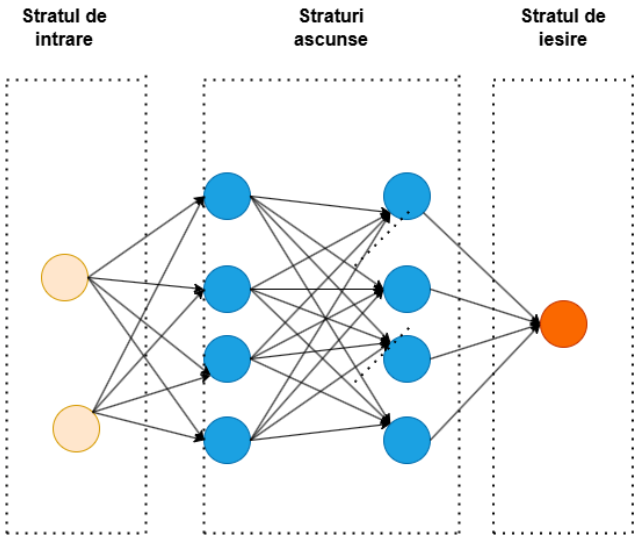


Figura 2.3 Schema unei rețele neuronale.

Fully Connected Layers (Dense Layers)

Straturile *Dense*, sau complet conectate, reprezintă cea mai simplă arhitectură de rețea neuronală, în care fiecare neuron dintr-un strat este conectat la toți neuronii din stratul anterior. Aceste straturi joacă un rol important în multe probleme de clasificare și regresie, deoarece permit modelarea relațiilor globale între caracteristici. O transformare tipică într-un astfel de strat este de forma y = f(Wx + b), unde f este o funcție de activare. Pentru a reduce complexitatea acestor tipuri de straturi, care necesită stocarea unei ponderi, au fost dezvoltate tehnici precum normalizarea lotului (*Batch Normalization*), abandonarea conexiunilor (*dropout*) și utilizarea unor funcții de activare.

Normalizarea lotului (*Batch normalization*)

Normalizarea lotului este o tehnică de regularizare care are rolul de a stabiliza și de a accelera procesul de antrenare prin normalizarea activărilor fiecărui strat. Aceasta contribuie la menținerea unei distribuții constante a datelor în timpul antrenării, reducând variațiile bruște între straturi (***internal covariate shift***) [14, pg. 2].

Rezultatul este o convergență mai rapidă, o generalizare mai bună și o reducere a riscului de supraînvățare. În practică, acest lucru permite utilizarea unor rate de învățare mai mari, cu un număr de epoci mai mic și face ca modelul să fie mai eficient în sarcini complexe, precum detectarea intruziunilor.

Dropout

Dropout este o metodă de regularizare ce reduce supraînvățarea în rețelele neuronale, prin dezactivarea aleatorie a neuronilor în timpul antrenării. Astfel, modelul învață să nu depindă excesiv de anumiți neuroni și devine mai robust, fiind capabil să generalizeze mai bine. Tehnica constă în ignorarea (cu o anumită probabilitate) a unor neuroni la fiecare pas din antrenare, forțând astfel rețeaua să găsească mai multe reprezentări utile pentru o sarcină, utilă în contexte precum detectarea anomaliilor, unde generalizarea este esențială pentru a identifica tipuri noi de atacuri. Deși poate fi utilizat împreună cu *Batch Normalization,* în unele cazuri poate deveni redundant [15].

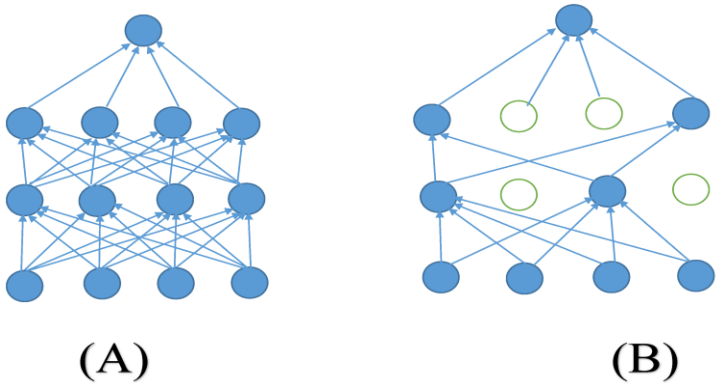


Figura 2.4 Ilustrarea unei rețele neuronale înainte de aplicarea *Dropout* (A) și după aplicare (B).

Funcțiile de activare sunt o componentă critică, cu un impact semnificativ asupra performanței modelului, care introduc neliniaritatea necesară pentru aproximarea funcțiilor complexe. Fără aceste funcții, o rețea neuronală, chiar și o rețea neuronală profundă, ar fi echivalentă cu o transformare liniară, limitând capacitatea de a învăța relațiile complexe dintre date [16, pg. 251].

Tipurile funcțiilor de activare:

1. **ReLU (Rectified Linear Unit)** – este cea mai utilizată funcție de activare în straturile ascunse, datorită simplității sale. Transformă valorile negative în 0 și păstrează valorile pozitive. Este eficientă din punct de vedere computațional, dar poate duce la inactivarea neuronilor, din cauza gradientului care devine zero pentru valorile negative [4, pg. 335].

2. **Leaky ReLU** – este o variantă a ReLU, care permite un gradient mic pentru valorile negative, prevenind astfel blocarea neuronilor.

3. **Softmax** – folosită în stratul de ieșire pentru clasificarea multi-clasă, transformând scorurile în probabilități. Este optimă împreună cu **funcția de pierdere a entropiei încrucișate** (*categorical_crossentropy*), reducând problemele cu optimizarea funcțiilor pătratice [16, pg. 252], penalizând modelul pentru predicții incerte [4, pg. 149]. Această funcție de pierdere măsoară diferența dintre distribuția de probabilitate a modelului și distribuția reală a claselor, ghidând modelul către o îmbunătățire a performanței [4, pg. 150].

Formulele detaliate pentru aceste funcții de activare sunt prezentate în Anexa G, formulele F.6–F.8.

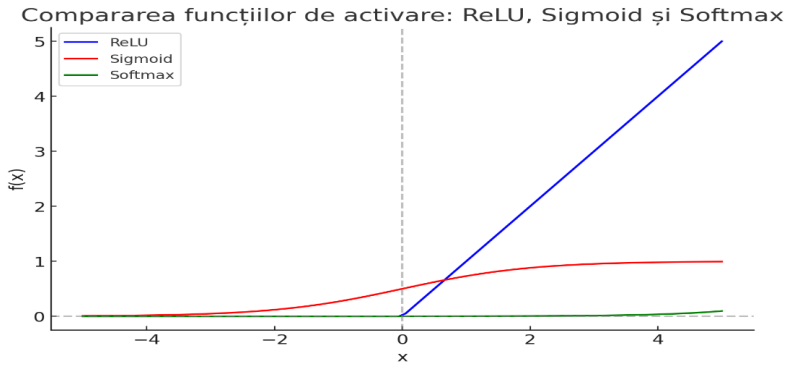


Figura 2.5. Se observă cum ReLU este liniară pentru valori ale lui x pozitive și zero pentru cele negative, Sigmoid normalizează ieșirile între 0 și 1, iar Softmax transformă scorurile în probabilități.

2.2. Metrici de evaluare a performanței

Evaluarea performanței modelelor de învățare automată utilizate este esențială pentru compararea metodelor și determinarea eficienței fiecărui model. În cazul clasificării traficului de rețea, metricile de evaluare permit compararea performanțelor modelelor utilizate pentru fiecare clasă de trafic din setul de date utilizat. În acest context, cele mai relevante metrici sunt **precizia** (*precision*), **recall-ul** (sensibilitatea), **acuratețea**, **F1-score-ul** și **matricea de confuzie**.

Precizia (*Precision*)

Precizia reprezintă raportul dintre numărul de instanțe clasificate corect pozitive și numărul total de instanțe clasificate pozitiv. Aceasta indică în ce măsură predicțiile de tip „atac” sunt corecte. O precizie mare va indica un număr mic de clasificări greșite, alarme false [4, pg. 91].

$$Precision = \frac{TP}{TP + FP} \qquad (2.1)$$

Unde:

TP (***True Positives***) – numărul de atacuri detectate corect.

FP (***False Positives***) – numărul de instanțe etichetate greșit.

Sensibilitatea (*Recall*)

Recall măsoară capacitatea modelului de a detecta corect atacurile. Calculează procentul atacurilor corect detectate din totalul de atacuri. Formula pentru recall este următoarea:

$$Recall = \frac{TP}{TP + FN} \quad (2.2)$$

Unde **FN** (***False Negatives***) reprezintă numărul de atacuri nedetectate de model.

Un *recall* scăzut indică faptul că multe atacuri nu sunt detectate. Pe de altă parte, un recall foarte mare va însemna că toate atacurile sunt detectate, dar vor exista alarme false, deoarece *precision* va scădea semnificativ. Există un compromis între *precision* și *recall*. Spre exemplu, un clasificator cu *precision* 30%, dar *recall* ridicat de 99%, este în regulă de folosit în detectarea atacurilor din rețea – aproape toate atacurile fiind detectate, dar vor exista alarme false [4, pg. 93].

F1-Score

F1-Score reprezintă un mod mai realist de a compara și evalua clasificatoarele. Se calculează ca media armonică dintre *precision* și *recall*, acordând echilibru celor două metrici, dar penalizând valorile mici [4, pg. 92]. Astfel, un model care are un *recall* sau o precizie mică va avea automat un F1-Score scăzut. Acesta este util în evaluarea clasificatorilor pe seturi de date dezechilibrate.

$$F1 = 2\,x\,\frac{Precision\ x\ Recall}{Precision + Recall} \qquad (2.3)$$

Acuratețea (*Accuracy*)

Semnificația acestui parametru constă în raportul dintre numărul total de predicții corecte și numărul total de instanțe din setul de date. Totuși, nu este considerată relevantă în cazul seturilor de date dezechilibrate, unde unele clase sunt mai dominante decât altele, deoarece poate obține scoruri ridicate doar prin clasificarea majorității instanțelor în categoria dominantă [4, pg. 90].

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \qquad (2.4)$$

Unde :

TN (***True Negatives***) – instanțele normale clasificate corect.

Matricea de confuzie

Un mod mult mai bun de a evalua performanța unui clasificator este prin matricea de confuzie, având ca idee generală obținerea unui număr care reprezintă de câte ori instanțele clasei X au fost clasificate drept Y.

Forma unei matrici de confuzie :

Clasa Reala (R) / Clasa Prezisa (P)	P1	P2
R1	TP_{11}	FP_{12}

R2	FP_{21}	TP_{22}

- $\boldsymbol{TP_{ii}}$ (***True Pozitives***) – numărul de instanțe din clasa *i* care au fost clasificate corect ca fiind din clasa *i*.

- $\boldsymbol{FP_{ij}}$ (***False Pozitives***) – numărul de instanțe din clasa *j* care au fost prezise greșit ca fiind din clasa *i*.

3. Implementare și testare

3.1. Setul de date

Pentru implementarea și testarea modelelor de clasificare multi-clasă a traficului din rețea, a fost utilizat setul de date **CIC-IDS 2017** (*Canadian Institute for CyberSecurity – Intrusion Detection System 2017*), creat de către CIC la Universitatea din New Brunswick. Datele conțin opt sesiuni de monitorizare a traficului, care au fost colectate pe parcursul a cinci zile (3 iulie – 7 iulie 2017), pe o rețea de calculatoare cu sisteme de operare moderne, inclusiv Windows 7 / 8.1 / 10, MAC, Ubuntu și Kali [18].

Setul de date conține următoarele tipuri principale de atacuri: DoS și DDoS (*Denial of Service* și *Distributed Denial of Service*), Botnet și diverse tipuri de *Brute Force*.

Setul de date CIC-IDS2017 conține de asemenea o varietate de caracteristici complexe. Spre deosebire de NSL-KDD, care nu include metrici relevante pentru analiza avansată a atacurilor, CIC-IDS2017 oferă **80 de caracteristici**, extrase cu **CICFlowMeter**. Dintre aceste caracteristici, unele au o relevanță foarte mare în identificarea unor tipuri de atacuri. De exemplu, „**SubFlow Fwd Bytes**” și „**Total Length Fwd Package**” sunt utile pentru detecția Botnet-urilor, „**Bwd Packet Length Std**” ajută la identificarea atacurilor DoS și DDoS, iar „**Init Win Fwd Bytes**” este relevant pentru atacurile *Brute Force*. Totodată, caracteristici precum „**Min Bwd Package Length**” și „**Fwd Average Package Length**” ajută la delimitarea

traficului legitim de cel malițios.

După cum se menționează în lucrările [18, 19], un alt avantaj al setului de date CIC-IDS2017 este faptul că datele sunt complet etichetate, fiind posibilă aplicarea directă a algoritmilor fără preprocesarea manuală a claselor. De asemenea, setul de date este disponibil fie sub formă de fișiere brute, în format *.pcap* (*packet capture*), care conțin capturile reale de trafic, dar și la versiuni preprocesate în format *.csv* (*Comma Separated Values*). Setul acoperă o gamă largă de protocoale precum HTTPS, SSH, FTP, ceea ce îl face mai variat în comparație cu alte seturi mai vechi. În plus, selecția tipurilor de atacuri simulate a fost realizată pe baza raportului de securitate McAfee din 2016, fapt ce contribuie la abundența și relevanța setului. Toate aceste caracteristici recomandă CIC-IDS2017 drept un set de date complet, complex și util pentru dezvoltarea sistemelor de detecție a intruziunilor.

3.2. Preprocesarea datelor

Pentru a putea utiliza setul de date în practică, este nevoie să fie aplicate modificări pentru eliminarea erorilor, standardizarea valorilor și conversia datelor în format adecvat.

În setul de date au mai fost identificate și corectate mai multe probleme. Coloana care definește direcția fluxului înainte (*forward*) apărea de două ori (coloana 41 și 62), iar pentru a elimina redundanța am eliminat coloana 62. Valori invalide precum **NaN** și **Infinity**, prezente în coloane precum „Flow Bytes/s”, „Flow Packet/s”, au fost înlocuite cu 0, respectiv 1, pentru a standardiza setul de date. De asemenea, la etichetele atacurilor web, a fost descoperit un caracter minus incompatibil (Unicode 8211), neputând fi interpretat corect de către Pandas, care folosește UTF (*Unicode Transformation Format*)-8, acesta fiind înlocuit cu Unicode 45, pentru a se asigura compatibilitatea cu restul setului.

Nu în ultimul rând, pentru a asigura compatibilitatea cu modelele de învățare automată, atributele precum *Source IP, Flow ID,* au fost convertite în valori numerice folosindu-se *Label Encoder* din biblioteca **Sklearn** [20]. Eticheta „Label” rămâne sub formă de variabilă categorială, fiind necesară pentru clasificarea multi-clasă a traficului. Distribuția finală a claselor după aceste transformări poate fi observată în Tabelul 1 din Anexa E.

3.3. Crearea seturilor de antrenare si test

Setul de date CIC-IDS2017 nu conține date separate pentru antrenare și testare, ci este furnizat întreg setul de date unificat. Pentru a evalua performanța modelelor, datele au fost împărțite în două subseturi distincte: set de antrenare (70% din întreg setul de date), respectiv set de testare (30% din setul de date original). Stabilirea procentelor de împărțire a setului de date depinde și de dimensiunea setului de date; spre exemplu, dacă am avea un set de date cu 10 milioane de instanțe, 1% ar fi de ajuns pentru setul de testare, distribuirea în proporțiile de 70%-30% fiind o practică comună.

Împărțirea setului de date se face folosind funcția **train_test_split** din biblioteca **Scikit-learn**, care asigură o împărțire aleatoare a datelor, garantând că modelul învață pe un subset diferit față de cel pe care este testat.

Pentru o evaluare mai precisă a performanței, am utilizat **Validarea Încrucișată Stratificată** (***Stratified Cross-Validation***), cu metoda **StratifiedKFol**d, deoarece permite testarea modelului pe multiple subseturi ale setului de date de antrenare. Am ales valoarea 3, împărțind astfel setul de date în 3 subseturi (fold-uri) egale. Astfel, la fiecare iterație, două fold-uri vor fi folosite pentru antrenare, iar unul pentru validare. Acest proces se repetă până când fiecare subset este folosit o singură dată pentru validare, scorul final fiind calculat ca media rezultatelor obținute. Astfel, validarea încrucișată oferă o estimare realistă a performanței modelului pe întreg setul de date de antrenament, contribuind astfel la îmbunătățirea generalizării modelului și minimizarea riscului de *overfitting* [21].

3.4. Selecția caracteristicilor relevante

Procesul de selecție a caracteristicilor (*feature selection*) are un rol esențial în antrenarea algoritmilor de învățare automată, deoarece nu toate caracteristicile au aceeași relevanță în clasificarea corectă a traficului. Rolul selecției caracteristicilor este de a elimina variabilele redundante, neesențiale, păstrându-le în final doar pe acelea care contribuie cel mai semnificativ la performanța modelului, aducând beneficii precum prevenirea riscului de *overfitting* [4, pg. 28], întrucât un număr mare de caracteristici poate introduce zgomot, făcând

modelul să învețe particularitățile setului de antrenare, devenind incapabil să aibă aceeași performanță și pe un set de date nou, incapabil să poată generaliza [22], precum și optimizarea timpului [4, pg. 198], deoarece modelele antrenate pe subseturi restrânse de caracteristici necesită mai puține resurse, accelerând astfel antrenarea algoritmilor.

3.4.1. Selecția caracteristicilor pentru Random Forest

Pentru modelul **Random Forest**, selecția caracteristicilor s-a realizat folosindu-se **importanța caracteristicilor** (***feature_importances***), metodă internă oferită de algoritm. Această metodă se bazează pe **indicele de impuritate Gini** și aplică tehnica **Mean Decrease in Impurity (MDI)**, despre care am discutat anterior.

După antrenarea modelului pe setul de date, caracteristicile au fost ordonate descrescător în funcție de importanța calculată (vezi Figura 1 din Anexa A). Apoi, au fost create mai multe subseturi de caracteristici (Top 10, 20, 30, 40), fiecare fiind testat separat, cu scopul de a găsi numărul optim de caracteristici care oferă echilibru între acuratețe și eficiență. Performanțele în funcție de numărul caracteristicilor selectate se pot vedea în Figura 2 din Anexa A, iar impactul selecției asupra performanței este ilustrat în Figura 3.

În concluzie, se observă că folosirea a 20–30 de caracteristici oferă cele mai bune rezultate, iar un număr mai mare nu crește semnificativ precizia, complexitatea modelului devenind foarte mare. Spre exemplu, se observă că trecerea de la folosirea a 20 de caracteristici la folosirea a 30 de caracteristici a contribuit cu o ușoară creștere a preciziei, în timp ce folosirea a 40 de caracteristici doar a crescut complexitatea modelului, dar a și scăzut performanțele. Astfel, pentru versiunea finală a modelului **Random Forest** de clasificare a traficului am ales folosirea primelor 20 de caracteristici, pentru a menține o generalizare eficientă și performanță ridicată.

3.4.2. Selecția caracteristicilor pentru XGBoost

În cadrul selecției caracteristicilor pentru antrenarea modelului **XGBoost**, a fost utilizată **metoda câștigului** (***Gain method*),** specifică algoritmului, despre care am vorbit anterior în capitolul dedicat XGBoost. Modelul a fost antrenat pe 70% din setul de date, importanța caracteristicilor este prezentată în Figura 1 din Anexa B.

Pe baza acestora, au fost create subseturi de câte 10 caracteristici (Top 10, 20, 30, 40), fiecare subset evaluat individual. Astfel, după cum se poate observa în Figurile 2 și 3 din Anexa B, se indică o îmbunătățire semnificativă și constantă a performanței pe măsură ce numărul caracteristicilor crește, în special *recall*-ul și scorul **F1**. Totuși, precizia scade ușor în ultimele subseturi, dar creșterea generală a performanței justifică alegerea unui număr mare de caracteristici.

În concluzie, deși din punct de vedere al complexității aduc un cost suplimentar, pentru antrenarea finală a modelului am ales folosirea primelor 40 de caracteristici, întrucât beneficiile pe care le aduce acest plus de informații justifică această creștere și contribuie la o clasificare mai eficientă.

3.5. Pregătirea datelor din setul de antrenament

Setul de date **CIC-IDS2017** este caracterizat printr-o distribuție foarte dezechilibrată a claselor. Spre exemplu, după cum se poate observa în Tabelul 1 din Anexa H, clasa „**Benign**” are peste 2,3 milioane de instanțe (aproximativ 81% din înregistrările totale), iar clase precum „**DDoS**”, „**Bot**” sau „**Infiltration**” au doar câteva zeci de instanțe. Acest dezechilibru poate cauza ca modelele de clasificare să favorizeze doar clasele dominante, ignorându-le pe celelalte, eșuând astfel în a detecta adecvat atacurile din rețea.

Pentru a aborda această problemă, voi utiliza tehnica de grupare a claselor, utilizarea tehnicii **SMOTE** (***Synthetic Minority Over-sampling Technique***) pentru clasele de dimensiune medie și *oversampling* manual pentru clasele extrem de rare.

Pentru a reduce complexitatea și a îmbunătăți detecția, clasele similare (ex. „**Brute Force**", „**XSS**", „**SQL Injection**") au fost reunite sub eticheta **Web Attack**, iar cele foarte rare (ex. „**HeartBleed**", „**Infiltration**") sub eticheta **Rare Events**.

În ciuda grupării unor clase, în continuare se poate observa că distribuția este una inegală, fiind diferențe foarte mari de instanțe între acestea. Astfel, voi aplica metoda **SMOTE**, care generează instanțe sintetice prin interpolare între eșantioanele din clasele minoritare, sporind astfel diversitatea datelor [23, pg. 328]. Metoda este dovedită că are capacitatea de a crește performanța de clasificare a claselor din setul de date dezechilibrat CIC-IDS2017 [24].

Funcționarea algoritmului SMOTE presupune, în primul rând, selectarea unei instanțe: pentru fiecare instanță din clasele minoritare, algoritmul identifică cei mai apropiați *k* vecini dintre celelalte instanțe minoritare. Ulterior, este selectat aleator unul dintre acești *k* vecini, iar un punct nou de date sintetic este creat pe segmentul de dreaptă care leagă instanța originală de vecinul ales. Procesul este repetat până când clasa minoritară a fost supra-eșantionată cu suficiente exemple pentru a se atinge echilibrul dorit al claselor.

Astfel, SMOTE va genera noi exemple de instanțe, similare cu cele existente deja, dar nu copii exacte. Chiar dacă cantitatea de eșantioane minoritare crește, calitatea este menținută, nefiind introdus zgomot aleator, algoritmul neschimbând forma generală a distribuției, ci doar face regiunile minoritare să devină mai dese [23, pg. 352].

Pentru clasele cu un număr moderat de instanțe (între 1.000 și 100.000), am aplicat SMOTE cu o rată de supra-eșantionare de 50%, iar pentru clasele extrem de rare (sub 1.000 de instanțe), unde interpolarea ar fi ineficientă din cauza numărului insuficient de vecini, am aplicat un *resampling* aleatoriu cu înlocuire, replicând fiecare instanță de 3 ori. Codul corespunzător acestei implementări este prezentat în Anexa C, Figura 2. Figura 3 din Anexa C evidențiază o creștere semnificativă a instanțelor claselor minoritare și rare, reducând dezechilibrul inițial. De asemenea, clasele moderate au avut o creștere controlată a instanțelor sintetice, însă diferențele față de clasele dominante rămân vizibile.

3.5.1 Implementarea Random Forest

În această secțiune voi detalia implementarea algoritmului, care va folosi caracteristicile relevante selectate în capitolul anterior (3.4.1). Pentru implementare, a fost utilizată biblioteca Scikit-learn din Python, care oferă clasa RandomForestClassifier. Inițial, modelul va fi creat cu setările implicite pentru a găsi combinația cea mai potrivită pentru hiperparametrii menționați la capitolul 3.4.1, specifici algoritmului. Pentru optimizarea acestora, am ales metoda **GridSearch**, întrucât numărul de parametri nu este prea mare și permite astfel explorarea tuturor combinațiilor posibile, evitând omiterea unei configurații optime [8]. De asemenea, permite integrarea cu **validarea încrucișată stratificată**, fiecare combinație de parametri fiind evaluată robust.

În urma execuției secvenței de cod prezentat în figura 1 din Anexa D, cu o durată de rulare de aproximativ 14 ore, am obținut următoarele valori optime ale hiperparametrilor: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 500}.

În evaluarea hiperparametrilor, am folosit metrica de evaluare **F1-Score weighted**, deoarece ia în considerare distribuția dezechilibrată a claselor și echilibrează impactul claselor minoritare. Dacă am fi utilizat doar acuratețea, modelul ar fi favorizat clasa majoritară (Benign), ignorând astfel clasele minoritare în capacitatea de clasificare. După cum se poate observa în Figura 2 din Anexa D, performanța crește simultan cu numărul de arbori, însă după 500 de arbori, îmbunătățirile devin neglijabile. De asemenea, se mai observă din Figura 3 a Anexei C faptul că adâncimea maximă None a permis arborilor să învețe mai multe detalii relevante, fără a introduce overfitting, având cele mai bune rezultate.

Cu această combinație de hiperparametri, va fi antrenat modelul final de clasificare, a cărei performanță va fi evaluată prin următoarele metrici: precizie, recall, scorul F1 și acuratețe.

După cum se poate observa în figurile 4 și 5 din Anexa D, modelul a reușit să detecteze majoritatea atacurilor, având un recall mediu ridicat de 88% și o precizie medie de 94%, ceea ce înseamnă un echilibru bun în identificarea corectă a atacurilor și minimizarea alertelor false. Totuși, atacurile rare „Rare Events” au fost dificil de identificat, din cauza numărului redus de instanțe din setul de antrenare (doar 7), fiind aproape complet ignorate.

În concluzie, Random Forest s-a dovedit un model robust și eficient pentru clasificarea traficului de rețea, dar pentru o detecție îmbunătățită a atacurilor rare (clasa Rare Events) și a atacurilor DDoS, poate fi abordată implementarea unui model mai complex precum XGBoost.

3.5.2. Implementarea XGBoost

Acest subcapitol se concentrează pe implementarea modelului XGBoost pentru clasificarea traficului. Vor fi detaliate configurarea inițială a modelului, procesul de optimizare al **hiperparametrilor** și evaluarea performanței finale. Caracteristicile pe care modelul le va **folosi**, sunt primele 40 cele mai relevante, despre care s-a discutat în capitolul 3.4.2.

Pentru modelul XGBoost va fi folosită biblioteca *xgboost* din Python. Optimizarea **hiperparametrilor** va fi realizată folosindu-se framework-ul **Optuna** din cauza numărului mare de **hiperparametri** ai modelului, făcând această sarcină aproape imposibilă pentru metodele clasice precum GridSearch din cauza timpului de rulare foarte mare [11]. Optuna folosește optimizarea **Bayesiană** pentru a găsi cei mai potriviți **hiperparametri** pentru model.

Optimizarea **Bayesiană** este asemănată unui **vânător de comori** care folosește un detector avansat de metale pentru a găsi aurul, în loc de a săpa gropi în toată zona (GridSearch) [11]. Optuna are nevoie de o funcție denumită **obiectiv** (*objective*), care va returna o valoare numerică pentru a evalua performanța **hiperparametrilor** și pentru a decide unde să eșantioneze în următoarele încercări [25]. Pentru funcția **obiectiv**, alegem să optimizăm **numărul de arbori**, **adâncimea maximă** a arborilor, **rata de învățare** și **factorii de regularizare L2 și L1**, cu un interval de valori ajustat corespunzător [25].

Din implementarea funcției obiectiv (Anexa E), se poate observa că este folosită **validarea încrucișată stratificată**, cu 3 fold-uri pentru o generalizare mai bună și metrica de evaluare **F1 score-weighted**, similar cu implementarea Random Forest din capitolul 3.5.1.

Obiectivul modelului este setat la valoarea **„multi:softprob”**, deoarece modelul trebuie să realizeze clasificări multi-clasă, mai multe tipuri de atac prezente în setul de date, unde modelul va returna probabilitățile pentru fiecare clasă. De asemenea, am ales metrica de evaluare **„mlogloss”**, deoarece penalizează predicțiile incorecte în funcție de probabilitățile asociate

claselor, potrivit pentru clasele dezechilibrate, scorurile obținute fiind influențate nu doar de corectitudine, dar mai mult de cât de sigure sunt predicțiile modelului. Astfel, optimizarea **hiperparametrilor** aleși a durat **53 de minute**, combinația optimă obținută fiind următoarea: { 'max_depth': 9, 'learning_rate': 0.1725242222887036, 'n_estimators': 392, 'reg_lambda': 0.039352461144354756, 'reg_alpha': 0.3699674307698651 }. Din Figura 2 (Anexa E) se observă că performanța modelului atinge un maxim în jurul valorii de 400 de arbori, mai exact la 392 de arbori. Totuși, în intervalul între 300 și 600, scorurile sunt foarte apropiate, ceea ce sugerează că un număr mai mare de arbori nu aduce o îmbunătățire semnificativă a performanței. De asemenea, din figura 3 (Anexa E), se mai observă faptul că scorul crește proporțional cu adâncimea maximă a arborilor, până la un interval optim de 7–9, apoi stabilizându-se sau scăzând ușor, observându-se că valoarea optimă a adâncimii este **7**.

În continuare, din cauza dezechilibrului claselor din setul de date, am ales să folosim o **funcție de pierdere ponderată (weighted loss function)** personalizată, a se observa în figura 4 din anexa E, care va aplica o penalizare mai mică pentru clasele minoritare, și o penalizare mai mare pentru clasele dominante. Astfel, modelul este împiedicat să se orienteze excesiv către clasa majoritară, putând învăța mai precis tiparele asociate claselor mai puțin frecvente.

În implementarea finală a modelului (figura 5 din anexa E), este utilizată tehnica **early_stopping** setată la 20 de runde. Pentru ca această tehnică să funcționeze, este nevoie să folosim **evals**, care verifică performanța pe setul de test (**x_test, y_test**) la fiecare iterație.

În concluzie, prin analiza performanței (a se analiza figurile 6 și 7 din Anexa E), **modelul XGBoost** se dovedește a fi superior celui Random Forest în ceea ce privește clasificarea traficului de rețea, aducând un echilibru mai bun între **Precizie** și **Recall**, în special pentru clasele **DDoS** și **Rare Events**, unde recall-ul a fost îmbunătățit. Deși a fost sacrificată o parte din precizie, modelul rămâne optim deoarece detectarea atacurilor este mai importantă decât reducerea alarmelor false.

Această îmbunătățire se datorează **funcției de pierdere ponderate** personalizate, dar și optimizării **hiperparametrilor** cu **Optuna** care a găsit combinațiile optime pentru a maximiza performanța într-un timp scurt și fără cost computațional prea mare, spre deosebire de **GridSearch-ul** folosit anterior la RandomForest.

3.6. Implementarea retelei neuronale

În acest subcapitol voi prezenta implementarea unei rețele neuronale artificială de tip ***fully connected (dense layers)***, folosind TensorFlow și API-ul (*Application Programming Interface*) Keras. Scopul este prezentarea fluxului de construire, metodele de preprocesare și setările cheie alese pentru a optimiza performanța.

Un prim pas important în optimizarea unei rețele neuronale îl reprezintă **standardizarea caracteristicilor**, deoarece ajută la menținerea distribuției stabile a ponderilor și la accelerarea procesului de antrenare [7, pg 398]. Această tehnică stabilizează gradientul în timpul antrenării, accelerând învățarea [16, pg 254], menținând parametrii într-o zonă numerică rezonabilă. Pentru realizarea acestui pas este folosit **StandardScaler** din biblioteca scikit-learn, transformând fiecare trăsătură astfel încât să aibă media 0 și abaterea standard 1 [26].

După scalarea datelor, etichetele vor fi transformate într-un format numeric, iar apoi se va aplica **one-hot encoding** (se observă figura 1 din Anexa F). Este o metodă standard în clasificarea multi-clasă, produce un vector binar care are lungimea egală cu numărul de clase și asigură compatibilitatea cu funcția de pierdere ***categorical_crossentropy***, fiind generat la ieșire un vector de probabilități, ușor de evaluat și interpretat [27].

Arhitectura finală constă într-un model **secvențial**, format din **trei straturi Dense**, care au 256, 128 respectiv 64 de neuroni, toți conectați între ei (Anexa F, Figura 2). Prima caracteristică folosită este **inițializarea He**, special optimizată pentru funcțiile de activare de tip ReLU (inclusiv **Leaky ReLU**), care are scopul principal de a menține distribuția fiecărui strat la nivel constant, asigurând convergența rețelelor foarte adânci [28, pg 3, 2.2]. Apoi, fiecărui strat dens i se aplică **BatchNormalization(),** pentru a normaliza activările înainte de aplicarea funcției de activare, pentru a preveni fluctuațiile excesive.

Se utilizează funcția **Leaky ReLU** cu **alpha=0.1**, variantă utilizată des în diverse lucrări precum [17], valoare validă dacă este testată și optimizată.

După cum se observă, primelor două straturi le este asociat un **Dropout** cu valorile de **0.4** respectiv **0.3**. Pentru primul strat cu 256 de neuroni, am aplicat un dropout ridicat de **0.4**,

deoarece are mulți neuroni și tinde să aibă o capacitate de memorare mare, putând astfel să ducă la *overfitting*, fără o regularizare mai puternică. Al doilea strat are o rată puțin mai mică, pentru a păstra mai mulți neuroni activi, informația fiind parțial filtrată de primul strat. Conform [15, pg 1933], un dropout de 20% în input layers și unul de 50% în hidden layers sunt valori comune pentru regularizarea optimă.

În final, stratul final este un strat **Dense** cu activare **softmax**, după care urmează compilarea modelului folosindu-se optimizatorul ***Adam (Adaptive Moment Estimation***) și funcția de pierdere ***categorical_crossentropy***. A fost ales optimizatorul **Adam**, deoarece are o capacitate foarte bună de adaptare, fiind o soluție optimă pentru majoritatea problemelor de *deep learning*. A fost aleasă valoarea **0.001** pentru rata de învățare, fiind o opțiune comună, recomandată pentru stabilitate și ușurință de utilizare [4, pg 357].

Pentru a îmbunătăți performanța modelului, am adăugat un **callback** (Figura 3 din anexa F) numit **ReduceLROnPlateau**, cu rolul de a ajusta automat rata de învățare în timpul antrenării. Acesta ține evidența pierderii pe setul de validare (*val_loss),* iar dacă nu observă îmbunătățiri timp de 3 epoci consecutive (*patience = 3*), va reduce rata de învățare la jumătate (*factor = 0.5*), modelul având o învățare mai fină.

În urma analizei rezultatelor din Figura 4 din Anexa F, se observă că performanța este foarte bună, modelul reușind să generalizeze bine pe setul de testare, inclusiv în cazul claselor rare, reușind să detecteze aproape toate instanțele malițioase.

3.7. Analiza comparativa a performantei

În această secțiune sunt analizate diferențele reale între cele trei modele testate, nu doar pe baza scorurilor generale, ci și în funcție de comportamentul fiecăruia pe clasele relevante pentru detecția traficului malițios. Deși toate modelele au avut performanțe bune pe clasele frecvente precum BENIGN sau DoS Hulk, diferențele au apărut clar la clasele rare sau dificil de detectat. **Rețeaua neuronală** a obținut cele mai bune rezultate generale, fiind singura care a atins un *recall* de 1.00 atât pe clasa Bot, cât și pe Web Attack, menținând totodată un scor foarte bun pe Rare Events (0.79).

XGBoost a avut performanțe apropiate, dar ușor mai slabe pe clase precum Bot și Web Attack. Totuși, se remarcă printr-un echilibru bun între precizie, timp de rulare și scoruri de clasificare. Modelul a fost optimizat folosind biblioteca *Optuna*, care a redus semnificativ timpul necesar reglării hiperparametrilor, obținând astfel rezultate solide într-un timp mult mai scurt comparativ cu alte metode. A fost singurul model care a atins același *recall* ca *Dense* pe clasa DDoS (0.90), demonstrând robustețe în fața atacurilor de volum.

Random Forest, deși performant în cazul claselor frecvente, a obținut rezultate semnificativ mai slabe pe clasele rare. Spre exemplu, pe Rare Events a obținut un *recall* de doar 0.29, față de 0.79 pentru celelalte două modele. În plus, procesul de optimizare prin *GridSearch* a durat peste 14 ore, fiind considerabil mai ineficient față de celelalte metode.

În concluzie, performanțele finale confirmă faptul că modelul de tip rețea neuronală este cel mai echilibrat din punct de vedere al detecției, în special pe clasele problematice. XGBoost rămâne o alternativă eficientă și bine optimizată, mai ales atunci când se dorește un compromis între precizie și timp de rulare, iar Random Forest poate fi potrivit doar pentru cazuri cu distribuție echilibrată sau când se prioritizează simplitatea modelului.

4. Concluzii

În urma testării celor trei modele de clasificare, pot spune că învățarea automată oferă soluții utile pentru identificarea traficului de rețea malițios. Rezultatele obținute mi-au permis să observ cum reacționează fiecare model în situații diferite, mai ales atunci când datele sunt dezechilibrate sau conțin clase rare.

Rețeaua neuronală a dat cele mai bune rezultate generale, fiind stabilă și precisă. **XGBoost** a fost foarte eficient din punct de vedere al timpului de rulare și a detectat bine atacurile mai greu de identificat. **Random Forest** s-a descurcat bine în cazul claselor mai frecvente, dar a avut nevoie de mai mult timp pentru a fi optimizat și nu a performat la fel de bine în cazurile rare. În plus, am observat că metoda de optimizare folosită are un impact clar asupra rezultatelor și timpului de rulare.

Prin lucrarea aceasta am reușit să construiesc un cadru clar și practic pentru testarea și

compararea mai multor algoritmi, cu pași bine definiți de prelucrare, antrenare și evaluare. Pe lângă rezultatele obținute, am înțeles mai bine cum trebuie adaptate metodele de învățare automată în funcție de datele pe care le ai și de problema pe care vrei să o rezolvi.

Tema poate fi dusă mai departe prin testarea altor modele mai simple sau mai rapide, precum *SVM* sau *KNN*, sau prin folosirea unor metode automate de alegere a caracteristicilor. De asemenea, ar fi interesant de urmărit cum se comportă modelele pe seturi de date mai mici sau pe date noi, pentru a vedea cât de bine se adaptează la alte situații.

BIBLIOGRAFIE

[1]. Cisco. Cisco Annual Internet Report (2018–2023) White Paper, https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html [accesat: 10.02.2025]

[2]. Z. Wang, K.-W. Fok, V. L. L. Thing. Machine Learning for Encrypted Malicious Traffic Detection: Approaches, Datasets and Comparative Study, Computers & Security, vol. 113, art. no. 102542, February 2022.

[3]. National Institute of Standards and Technology. Security and Privacy Controls for Information Systems and Organizations, NIST Special Publication 800-53 Revision 5, September 2020, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf [accesat: 03.04.2025].

[4]. A. Géron. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd ed. O'Reilly Media, 2019.

[5]. L. Breiman. Random Forests, Machine Learning, vol. 45, pp. 5–32, October 2001.

[6]. L. Breiman. Bagging Predictors, Machine Learning, vol. 24, pp. 123–140, August 1996.

[7]. T. Hastie, R. Tibshirani, J. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd ed. Springer, 2009.

[8]. N. Selvaraj. Hyperparameter Tuning Using Grid Search and Random Search in Python, KDnuggets, Oct. 2022, https://www.kdnuggets.com/2022/10/hyperparameter-tuning-grid-search-random-search-python.html [accesat: 06.12.2024]

[9]. J. P. Mueller, L. Massaron. Machine Learning For Dummies Cheat Sheet, Dummies.com, March 2021, https://www.dummies.com/article/technology/information-technology/ai/machine-learning/machine-learning-dummies-cheat-sheet-221432 [accesat: 07.12.2024]

[10]. T. Chen, C. Guestrin. XGBoost: A Scalable Tree Boosting System, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794, August 2016.

[11]. M. Filho. XGBoost Hyperparameter Tuning With Optuna (Kaggle Grandmaster Guide), Forecastegy, April 2023, https://forecastegy.com/posts/xgboost-hyperparameter-tuning-with-optuna [accesat: 12.12.2024].

[12]. Y. LeCun, Y. Bengio, G. Hinton. Deep Learning, Nature, vol. 521, no. 7553, pp. 436–444, 2015.

[13]. D. E. Rumelhart, G. E. Hinton, R. J. Williams. Learning representations by back-propagating errors, Nature, vol. 323, nr. 6088, pp. 533–536, October 1986.

[14]. S. Ioffe, C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Proceedings of the 32nd International Conference on Machine Learning, vol. 37, pp. 448–456, July 2015.

[15]. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Journal of Machine Learning Research, vol. 15, pp. 1929–1958, January 2014.

[16]. X. Glorot, Y. Bengio. Understanding the Difficulty of Training Deep Feedforward Neural Networks, Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, pp. 249–256, May 2010.

[17]. A. L. Maas, A. Y. Hannun, A. Y. Ng. Rectifier Nonlinearities Improve Neural Network Acoustic Models, Proceedings of the 30th International Conference on Machine Learning, vol. 28, part 3, pp. 1–6, 2013.

[18]. I. Sharafaldin, A. Gharib, A. H. Lashkari, A. Ghorbani. Towards a Reliable Intrusion Detection Benchmark Dataset, Software Networking, vol. 2017, pp. 177–200, 2017.

[19]. A. Gharib, I. Sharafaldin, A. H. Lashkari, A. A. Ghorbani. An Evaluation Framework for

Intrusion Detection Dataset, Proceedings of the International Conference on Information Science and Security (ICISS), pp. 1–6, 2016.

[20]. Scikit-learn. LabelEncoder — sklearn.preprocessing.LabelEncoder, https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html [accesat: 05.01.2025].

[21]. Scikit-learn. StratifiedKFold — sklearn.model_selection.StratifiedKFold, https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html [accesat: 05.01.2025].

[22]. Amazon Web Services. What is Overfitting?, https://aws.amazon.com/what-is/overfitting/ [accesat: 05.01.2025].

[23]. N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique, Journal of Artificial Intelligence Research, vol. 16, pp. 321–357, 2002.

[24]. A. A. Alfrhan, R. H. Alhusain, R. U. Khan. SMOTE: Class Imbalance Problem in Intrusion Detection System, Proceedings of the 2020 International Conference on Computing and Information Technology (ICCIT-1441), pp. 1–5, 2020.

[25]. C. Loomis. Using Optuna to Optimize XGBoost Hyperparameters, https://medium.com/optuna/using-optuna-to-optimize-xgboost-hyperparameters-63bfcdfd3407 [accesat: 09.01.2025]

[26]. Scikit-learn. StandardScaler — sklearn.preprocessing.StandardScaler, https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html [accesat: 09.01.2025]

[27]. F. Chollet. Deep Learning with Python, 2nd ed., Manning Publications, October 2021.

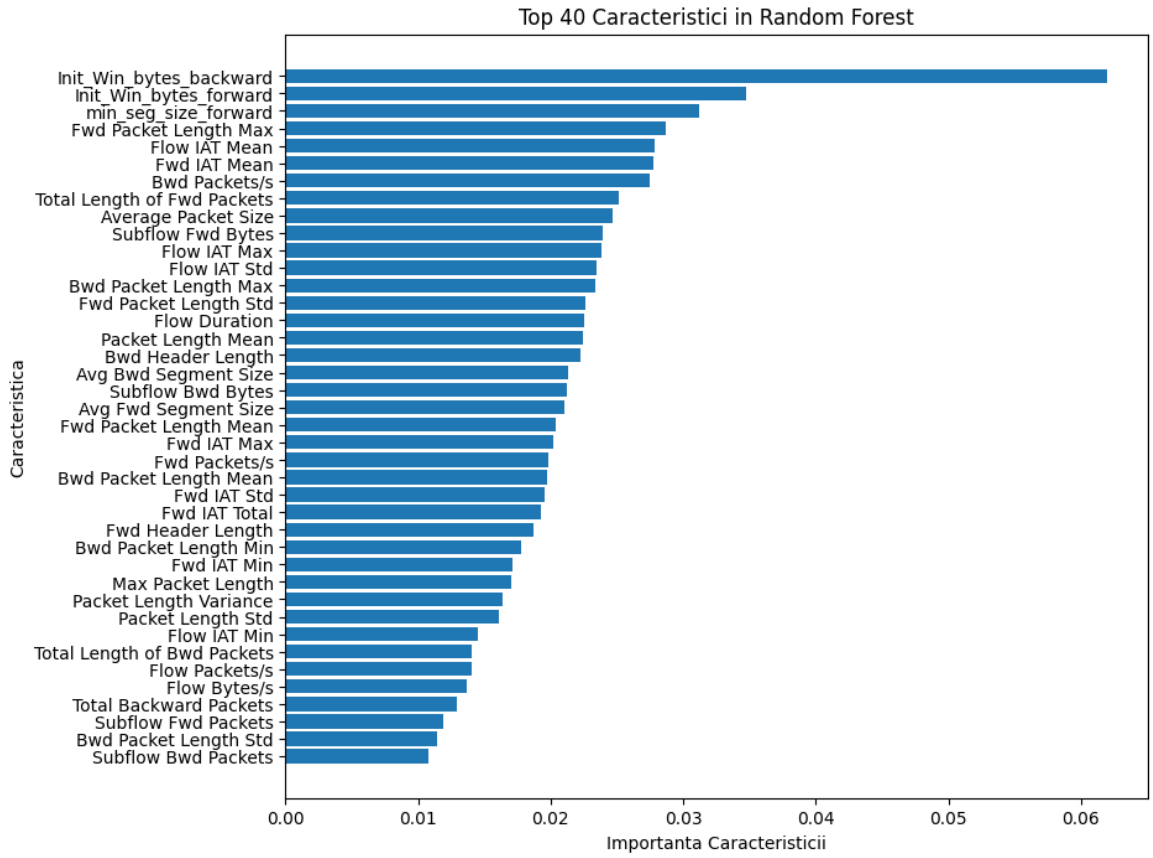
[28]. K. He, X. Zhang, S. Ren, J. Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), pp. 1026–1034, 2015.

[29]. https://www.researchgate.net/figure/The-structure-of-random-forest-RF-RF-is-one-of-the-bagging-bootstrap-aggregating_fig2_361835982 [accesat: 09.02.2025]

[30]. https://dzone.com/articles/xgboost-a-deep-dive-into-boosting [accesat: 10.02.2025]

Anexe

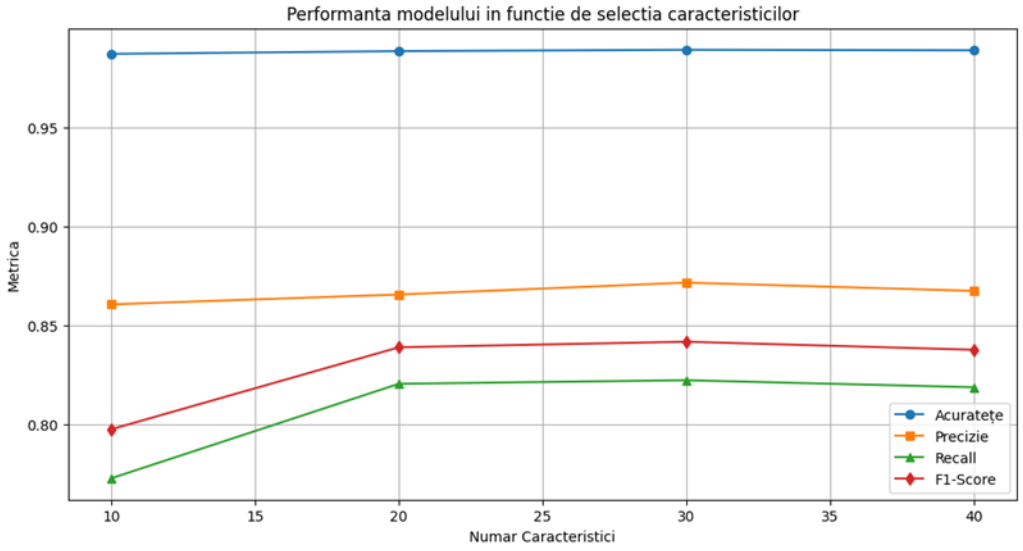
Anexa A – Importanța caracteristicilor pentru modelul Random Forest



Anexa A, Figura 1 – ilustrează importanța primelor 40 de caracteristici conform metodei Gain Importance.

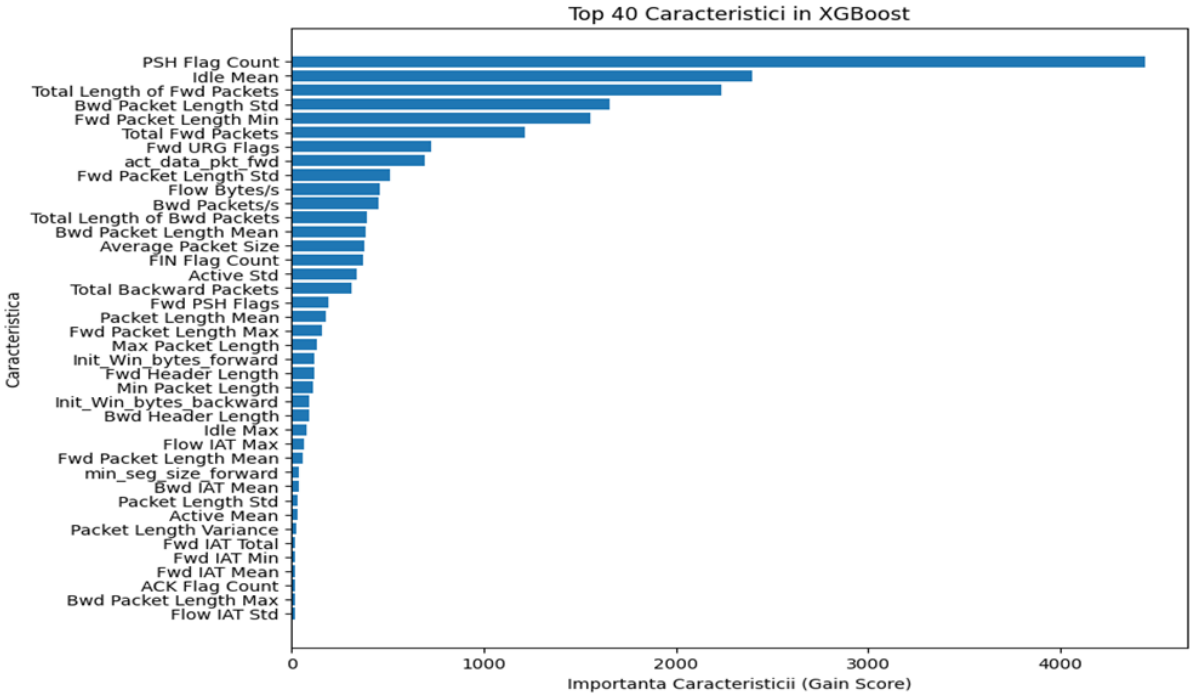
	Top Caracteristici	Acuratete	Precizie	Recall	F1-Score
0	10	0.986986	0.860705	0.773037	0.797744
1	20	0.988353	0.865661	0.820650	0.839047
2	30	0.989047	0.871660	0.822450	0.841862
3	40	0.988793	0.867493	0.818904	0.837804

Anexa A, Figura 2 – prezintă rezultatele obținute pentru fiecare subset de caracteristici, comparând acuratețea, precizia, recall-ul și scorul F1.



Anexa A, Figura 3 – se observă că, pentru subsetul alcătuit din Top 10 caracteristici, performanțele sunt scăzute din cauza informațiilor lipsă, apoi, crescând treptat numărul de caracteristici, modelul se stabilizează.

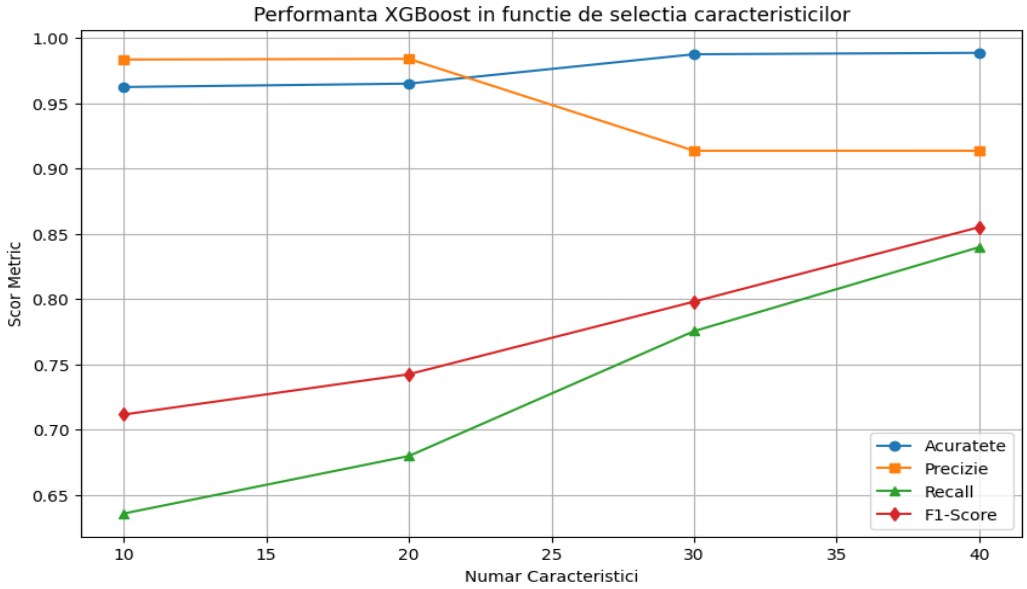
Anexa B – Importanța caracteristicilor pentru modelul XGBoost



Anexa B, Figura 1 – ilustrează importanța primelor 40 de caracteristici conform metodei Gain Importance.

	Top Caracteristici	Acuratete	Precizie	Recall	F1-Score
0	10	0.962314	0.983351	0.635844	0.711627
1	20	0.964907	0.983855	0.679869	0.742473
2	30	0.987385	0.913538	0.775432	0.798035
3	40	0.988475	0.913516	0.839695	0.855017

Anexa B, Figura 2 – prezintă rezultatele obținute pentru fiecare subset de caracteristici, comparând acuratețea, precizia, recall-ul și scorul F1.



Anexa B, Figura 3 – compară metricele de performanță ale modelului, observându-se că, pe măsură ce subsetul de caracteristici crește, cresc proporțional și metricele modelului.

Anexa C - SMOTE

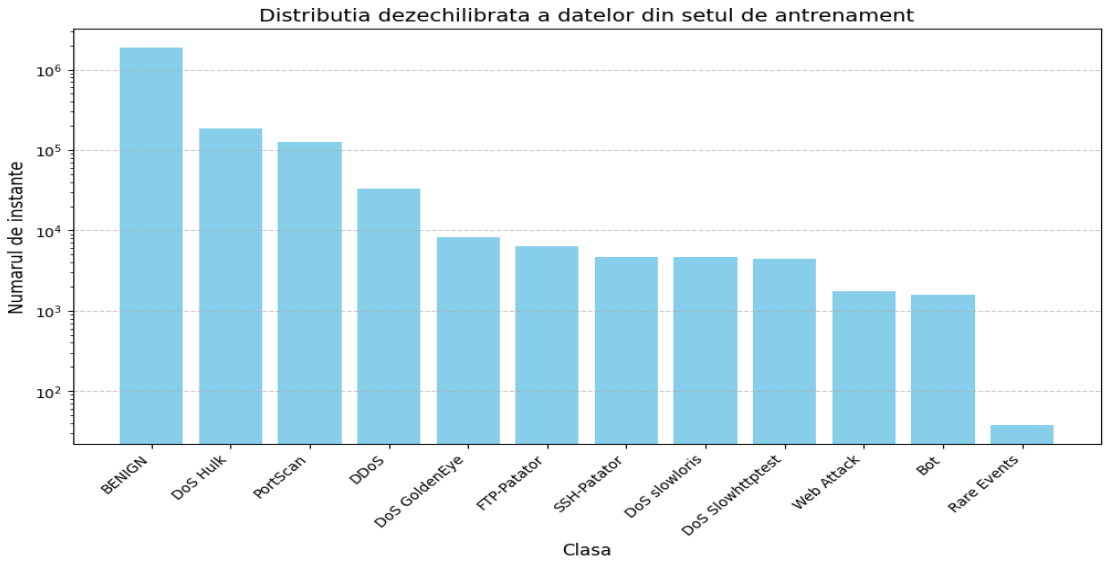


Figura 1 prezintă dezechilibrul prezent în setul de antrenament (70% din setul de date original) pe o scară logaritmică, subliniind nevoia de suplimentare a instanțelor din clasele minoritare.

```
moderate_classes = original_counts[(original_counts >= 1_000) & (original_counts < 100_000)].index
rare_classes = original_counts[original_counts < 1_000].index

smote = SMOTE(
    sampling_strategy={label: int(count * 1.5) for label, count in original_counts.items() if label in moderate_classes},
    random_state=42
)

X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

for label in rare_classes:
    class_data = X_train_smote[y_train_smote == label]
    class_labels = y_train_smote[y_train_smote == label]

    oversampled_data = resample(
        class_data,
        replace=True,
        n_samples=len(class_data) * 3,
        random_state=42
    )

    oversampled_labels = [label] * len(oversampled_data)

    X_train_smote = pd.concat([pd.DataFrame(X_train_smote), pd.DataFrame(oversampled_data)])
    y_train_smote = pd.concat([pd.Series(y_train_smote), pd.Series(oversampled_labels)])
```

Figura 2 prezintă tehnica SMOTE aplicată pe clasele cuprinse între 1.000 și 100.000 de instanțe, fiind supra-eșantionate prin adăugarea a 50% de instanțe sintetice. Clasele rare, care au mai puțin de 1.000 de instanțe, sunt replicate de 3 ori prin resampling cu înlocuire.

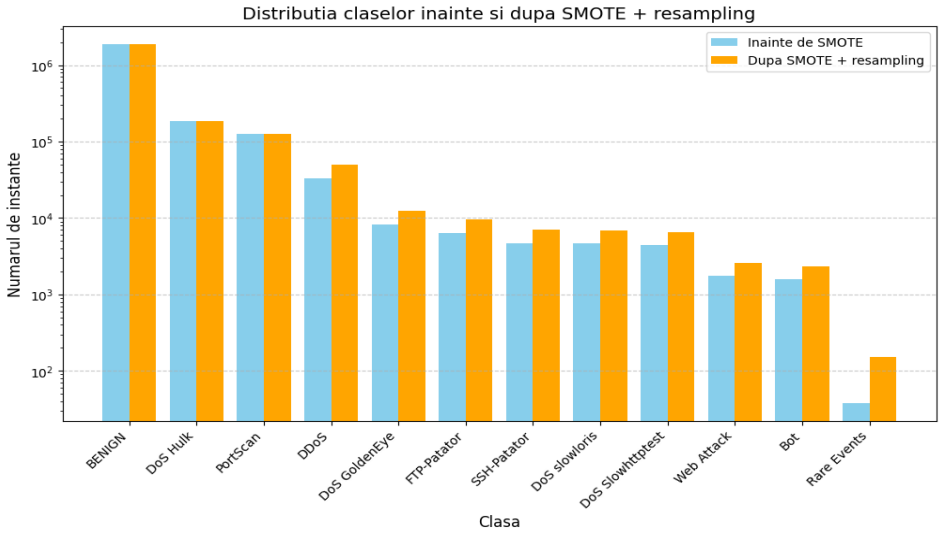
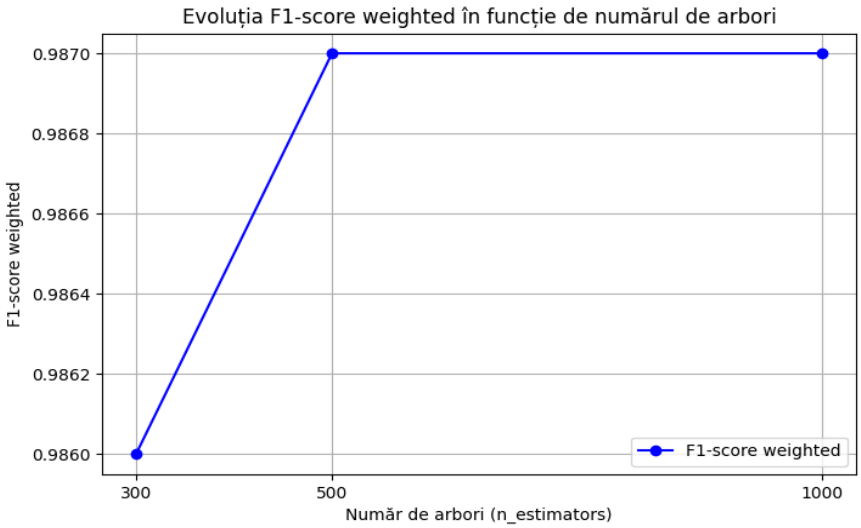


Figura 3 prezintă comparația distribuției claselor înainte și după aplicarea SMOTE și *resampling*-ului aleator.

Anexa D – Implementarea și evaluarea performanței modelului Random Forest

```
1  rf_base = RandomForestClassifier(class_weight="balanced", random_state=42, n_jobs=-
1)

2  param_grid = {
3      'n_estimators': [300, 500, 1000],
4      'max_depth': [20, 30, None],
5      'min_samples_split': [2, 5],
6      'min_samples_leaf': [1, 2]
7  }

8  cv_strategy = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

9  grid_search = GridSearchCV(rf_base, param_grid, cv=cv_strategy,
10                             scoring="f1_weighted", n_jobs=1, verbose=3)

11 start_time = time.time()
12 grid_search.fit(X_train_smote, y_train_smote)
13 print("\nGridSearch completat in %.2f secunde!" % (time.time() - start_time))

14 print("\nCei mai buni hiperparametri:", grid_search.best_params_)
```

Anexa D, Figura 1 – codul pentru inițializarea unui model Random Forest de bază asupra căruia testăm toate combinațiile posibile de hiperparametri pentru a găsi configurația optimă pentru seturile de date folosite.



Anexa D, Figura 2 – evoluția scorului F1 în funcție de numărul arborilor.

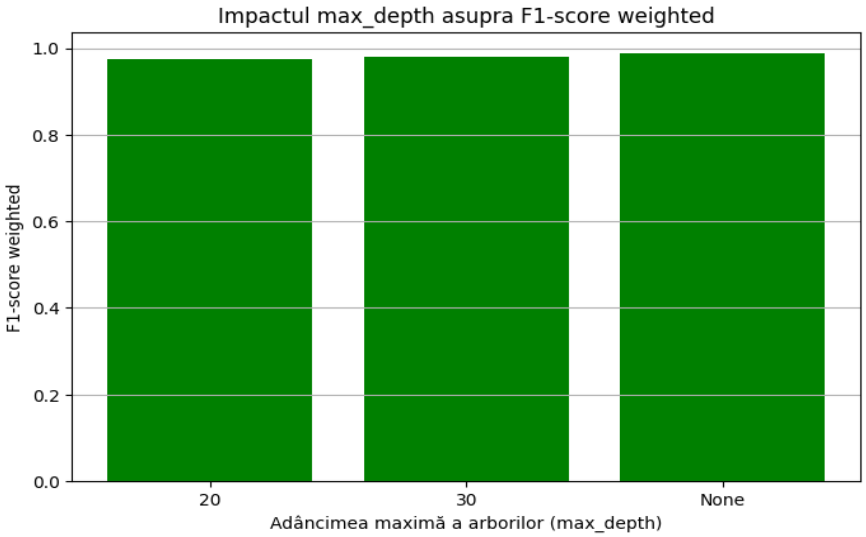
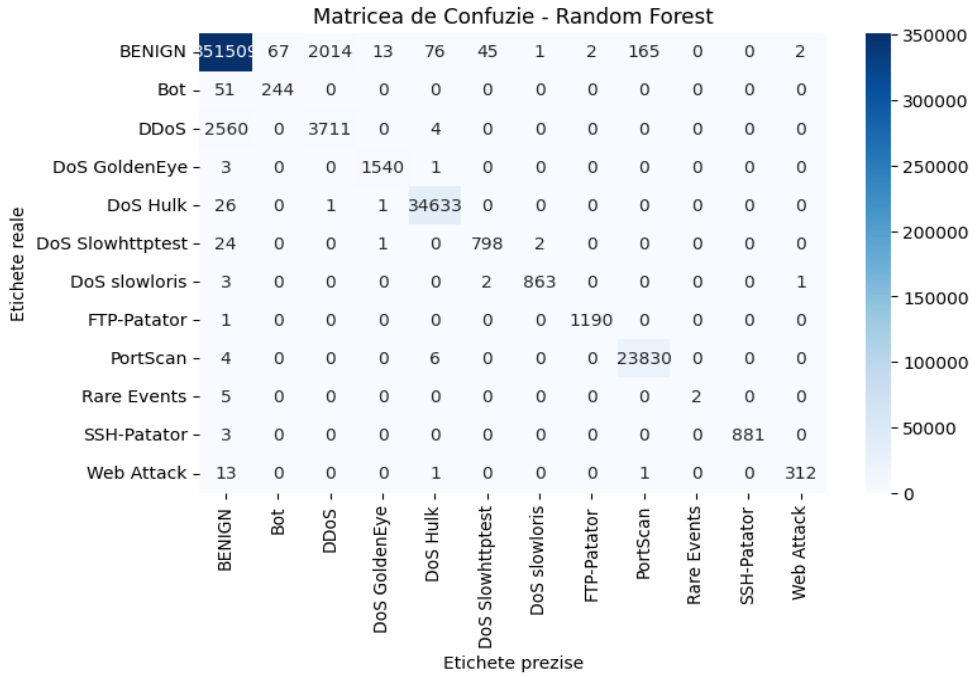


Figura 3, Anexa D – scorul F1 în funcție de adâncimea maximă.

Clasă	Precizie	Recall	F1-Score	Număr probe (support)
BENIGN	0.99	0.99	0.99	353894
Bot	0.78	0.83	0.81	295
DDoS	0.65	0.59	0.62	6275
DoS GoldenEye	0.99	1.00	0.99	1544
DoS Hulk	1.00	1.00	1.00	34661
DoS Slowhttptest	0.94	0.97	0.96	825
DoS slowloris	1.00	0.99	0.99	869
FTP-Patator	1.00	1.00	1.00	1191
PortScan	0.99	1.00	1.00	23840
Rare Events	1.00	0.29	0.44	7
SSH-Patator	1.00	1.00	1.00	884
Web Attack	0.99	0.95	0.97	327
Accuracy	NaN	0.99	NaN	424612
Macro Avg	0.94	0.88	0.90	424612
Weighted Avg	0.99	0.99	0.99	424612

Anexa D, Figura 4 – rezultatele modelului Random Forest antrenat cu configurația hiperparametrilor identificată de către GridSearch.

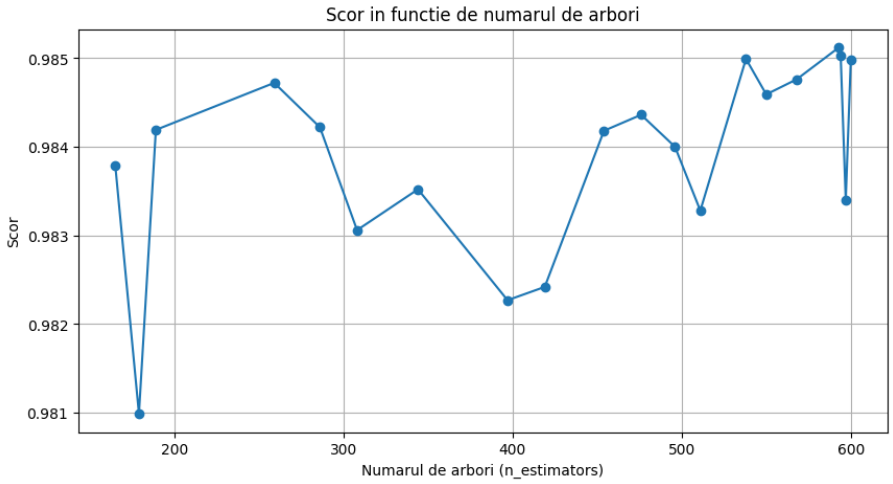


Anexa D, Figura 5 – matricea de confuzie a modelului Random Forest. Prezintă numărul de instanțe clasificate corect (diagonala principală) și erorile de clasificare pentru fiecare clasă (în afara diagonalei).

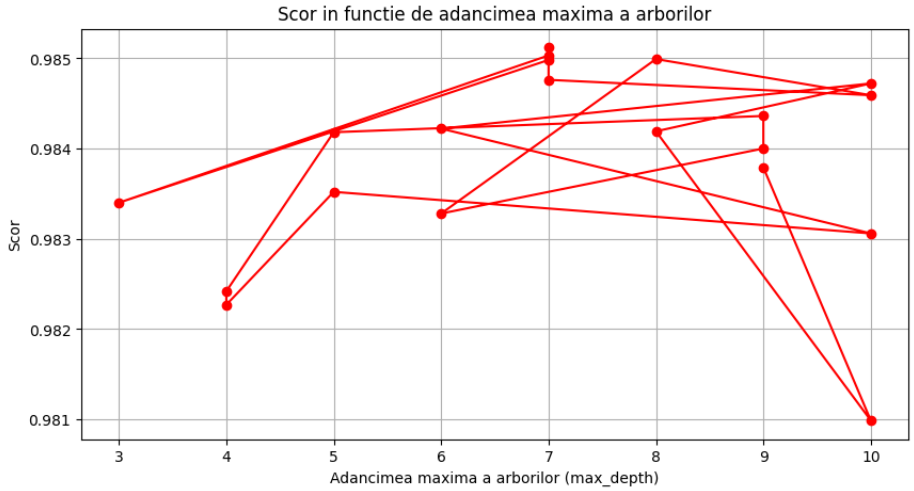
Anexa E

```
def objective(trial):
    max_depth = trial.suggest_int("max_depth", 3, 10)
    learning_rate = trial.suggest_float("learning_rate", 0.01, 0.2)
    n_estimators = trial.suggest_int("n_estimators", 150, 600)
    reg_lambda = trial.suggest_float("reg_lambda", 0.01, 10)  # Regularizare L2
    reg_alpha = trial.suggest_float("reg_alpha", 0.0, 1.0)    # Regularizare L1
    subsample = trial.suggest_float("subsample", 0.05, 1.0)
    colsample_bytree = trial.suggest_float("colsample_bytree", 0.05, 1.0)

    model = XGBClassifier(
        objective="multi:softprob",
        num_class=len(balanced_counts),
        max_depth=max_depth,
        learning_rate=learning_rate,
        n_estimators=n_estimators,
        subsample=subsample,
        colsample_bytree=colsample_bytree,
        random_state=42,
        eval_metric="mlogloss",
        reg_lambda=reg_lambda,
        reg_alpha=reg_alpha,
        tree_method="gpu_hist",
        gpu_id=0
    )

    cv_strategy = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
    scores = cross_val_score(model, X_train_smote, y_train_smote,
                             cv=cv_strategy, scoring="f1_weighted", n_jobs=-1)
    return np.mean(scores)
```

Anexa E, Figura 1 – codul funcției objective care explorează combinațiile hiperparametrilor pentru obținerea celei mai bune performanțe.



Anexa D, Figura 2 – Evoluția scorului F1 în funcție de numărul de arbori.



Anexa D, Figura 3 – Scorul F1 în funcție de adâncimea maximă a arborilor.

```
def weighted_loss(preds, dtrain):
    labels = dtrain.get_label()
    preds = preds.reshape(len(labels), -1)
    preds = np.exp(preds) / np.sum(np.exp(preds), axis=1, keepdims=True)

    # Reducem penalizarea pentru clasele rare
    class_1weights = {cls: (max(balanced_counts.values()) / count) ** 0.8
                      for cls, count in balanced_counts.items()}
    weight_array = np.array([class_weights[int(label)] for label in labels])

    grad = (preds - np.eye(preds.shape[1])[labels.astype(int)]) * weight_array.reshape(-1, 1)
    hess = preds * (1.0 - preds) * weight_array.reshape(-1, 1)

    return grad.flatten(), hess.flatten()
```

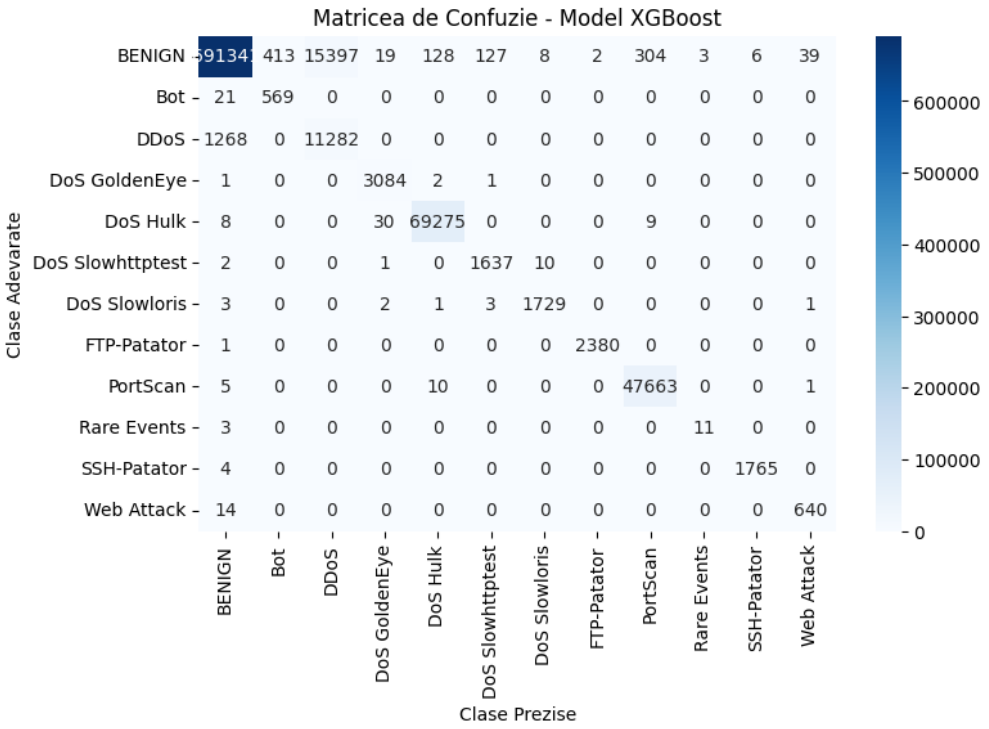
Anexa E, Figura 4 – implementarea funcției de pierdere ponderată, utilizată pentru XGBoost, cu exponentul 0.8 ales pentru a reduce penalizarea claselor rare, pentru a le acorda mai multă atenție și pentru a echilibra învățarea modelului.

```python
model = xgb.train(
    {
        "objective": "multi:softprob",
        "num_class": len(balanced_counts),
        "max_depth": best_params["max_depth"],
        "learning_rate": best_params["learning_rate"],
        "n_estimators": best_params["n_estimators"],
        "subsample": 0.9,
        "colsample_bytree": 0.9,
        "random_state": 42,
        "tree_method": "gpu_hist",
        "gpu_id": 0,
        "reg_lambda": best_params["reg_lambda"],   # Regularizare L2
        "reg_alpha": best_params["reg_alpha"]      # Regularizare L1
    },
    xgb.DMatrix(X_train_smote, label=y_train_smote),
    num_boost_round=best_params["n_estimators"],
    evals=[(xgb.DMatrix(X_test, label=y_test), "eval")],  # Validare in timpul antrenarii
    early_stopping_rounds=20,
    obj=weighted_loss
)
```

Anexa E, Figura 5 – antrenarea finală a modelului XGBoost cu hiperparametrii optimi identificați de către Optuna și folosirea funcției de pierdere ponderată.

	Clasa	Precizie	Recall	F1-Score	Numar Probe(Support)
0	BENIGN	1.0	0.98	0.99	707787
1	Bot	0.58	0.96	0.72	590
2	DDoS	0.42	0.90	0.58	12550
3	DoS GoldenEye	0.98	1.00	0.99	3088
4	DoS Hulk	1.0	1.00	1.0	69322
5	DoS Slowhttptest	0.93	0.99	0.96	1650
6	DoS slowloris	0.99	0.99	0.99	1739
7	FTP-Patator	1.0	1.00	1.0	2381
8	PortScan	0.99	1.00	1.0	47679
9	Rare Events	0.79	0.79	0.79	14
10	SSH-Patator	1.0	1.00	1.0	1769
11	Web Attack	0.94	0.98	0.96	654
12	accuracy	-	0.98	-	849223
13	macro avg	0.88	0.97	0.91	849223
14	weighted avg	0.99	0.98	0.98	849223

Anexa E, Figura 6 – performanța finală pe fiecare clasă a modelului XGBoost cu hiperparametrii optimi.



Anexa E, Figura 7 – matricea de confuzie obținută în urma antrenării modelului XGBoost.

Anexa F - Implementarea rețelei neuronale

```
# Scalarea datelor
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_smote)
X_test_scaled = scaler.transform(X_test)

# Transformarea etichetelor în format numeric
label_mapping = {label: idx for idx, label in enumerate(sorted(np.unique(y_train_smote)))}
y_train_numeric = np.array([label_mapping[label] for label in y_train_smote])
y_test_numeric = np.array([label_mapping[label] for label in y_test])

# Conversia la one-hot encoding
y_train_categorical = to_categorical(y_train_numeric)
y_test_categorical = to_categorical(y_test_numeric)
```

Anexa F, Figura 1 – implementarea scalării datelor, conversia etichetelor în format numeric și *one-hot encoding* pentru pregătirea datelor.

```
def create_optimized_model(input_shape, num_classes):
    model = Sequential()

    # Stratul 1: Dense + BatchNorm + LeakyReLU + Dropout
    model.add(Dense(256, input_shape=(input_shape,), kernel_initializer='he_normal'))
    model.add(BatchNormalization())
    model.add(tf.keras.layers.LeakyReLU(alpha=0.1))
    model.add(Dropout(0.4))

    # Stratul 2: Dense + BatchNorm + LeakyReLU + Dropout
    model.add(Dense(128, kernel_initializer='he_normal'))
    model.add(BatchNormalization())
    model.add(tf.keras.layers.LeakyReLU(alpha=0.1))
    model.add(Dropout(0.3))

    # Stratul 3: Dense + BatchNorm + LeakyReLU
    model.add(Dense(64, kernel_initializer='he_normal'))
    model.add(BatchNormalization())
    model.add(tf.keras.layers.LeakyReLU(alpha=0.1))

    # Stratul final: Softmax pentru clasificare multi-clasă
    model.add(Dense(num_classes, activation='softmax'))

    # Compilarea modelului
    optimizer = Adam(learning_rate=0.001)
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

    return model
```

Anexa F, Figura 2 – definirea arhitecturii rețelei neuronale.

	precision	recall	f1-score	support
BENIGN	1.00	0.99	0.99	707787
Bot	0.99	1.00	0.99	590
DDoS	0.64	0.90	0.75	12550
DoS GoldenEye	1.00	0.99	0.99	3088
DoS Hulk	1.00	1.00	1.00	69322
DoS Slowhttptest	0.99	0.99	0.99	1650
DoS slowloris	0.99	0.99	0.99	1739
FTP-Patator	1.00	1.00	1.00	2381
PortScan	1.00	1.00	1.00	47679
Rare Events	0.73	0.79	0.76	14
SSH-Patator	1.00	1.00	1.00	1769
Web Attack	0.99	1.00	1.00	654
accuracy			0.99	849223
macro avg	0.94	0.97	0.96	849223
weighted avg	0.99	0.99	0.99	849223

Anexa F, Figura 3 – implementarea metodei *ReduceLROnPlateau*.

Anexa G – Formule utile

Formula F.1 – Indicele Gini : $G = 1 - \sum_{i=1}^{C} p_i^2$, unde p_i este probabilitatea ca un exemplu să aparțină clasei i într-un anumit nod; o valoare mai mică a indicelui Gini semnalează o separare mai bună.

Formula F.2 – Entropia : $Entropia = -\sum_{i=1}^{C} p_i \log_2(p_i)$

Formula F.3 – Funcția obiectiv XGBoost [10, pg 2]:

$$L = \sum_{i=1}^{n} l(y_i, \hat{y_i}) \sum_{k=1}^{K} \Omega(f_k),$$

unde :

- $l(y_i, \hat{y_i})$ reprezintă funcția de pierdere, evaluând diferența între valoarea reală și cea prezisă.

- $\Omega(f_k)$ este termenul de regularizare, responsabil cu controlul complexității arborilor.

Formula F.4 – Termenul de regularizare XGBoost :

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda \sum_j w_j^2 + \alpha \sum_j |w_j| \quad \text{, unde :}$$

- T – numărul de noduri din arbore.
- w_j – ponderile atribuite frunzelor.
- $\gamma -$ parametru care penalizează numărul de noduri, previne arborii mari și complecși.
- λ (L2, Ridge) – util la modelele instabile, reduce sensibilitatea la zgomot, supraantrenare și stabilizează predicțiile.
- α (L1, Lasso) –util la date cu multe caracteristici, penalizează valoarea absolută a coeficienților, elimină caracteristicile irelevante reducând astfel dimensiunea modelului.

Formula F.5 – Scorul câștigului (Gain) în XGBoost:

$$\text{Gain} = \text{Loss}_{\text{parent}} - (\text{Loss}_{\text{left}} + \text{Loss}_{\text{right}}) \text{ , unde :}$$

$\text{Loss}_{\text{parent}}$ - pierderea totală înainte de împărțirea nodului.

$\text{Loss}_{\text{left}}$ si $\text{Loss}_{\text{right}}$ – pierderile după împărțire, pentru fiecare dintre cele două noduri copil.

Formula F.6 – Funcția ReLU : f (x) = max (0, x)

Formula F.7 – Funcția Leaky ReLU :

$$f(x) = \begin{cases} x, & dacă\ x > 0 \\ \alpha x, & dacă\ x \leq 0 \end{cases}$$

Alpha este un parametru mic (de exemplu 0.01), care evită neuronii inactivi, păstrând un grad mic de informație pe valorile negative [17, pg. 2].

Formula F.8 – Funcția Softmax : $f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$, unde :

- z_i reprezintă scorul brut pentru clasa i.
- K numărul total de clase.

Funcția asigură ca ieșirea este o distribuție de probabilitate (suma valorilor este 1).

Anexa H – Tabelul de distribuție al claselor.

Benign	2.359.289
DoS Hulk	231.073
PortScan	158.903
DDoS	41.835
DoS GoldenEye	10.293
FTP-Patator	7.938
SSH-Patator	5.897
DoS Slowloris	5.796
DoS Slowhttptest	5.499
Bot	1.966
Web Attack – Brute Force	1.507
Web Attack – XSS	652
Infiltration	36
Web Attack – SQL Injection	21
Heartbleed	11

Anexa H, Tabel 1. Distribuția claselor în setul de date.