Tema 2 - Machine Learning

Fichios Mara
342 C3
mara.fichios@stud.acs.upb.ro

Rezumat

Această lucrare abordează clasificarea sentimentului în limba română folosind un flux complet de prelucrare și modelare pentru date text. Am pornit de la o analiză exploratorie pentru a înțelege distribuția etichetelor, lungimea mesajelor și tiparele de zgomot (URL-uri, emoticoane, punctuație repetată, metadate). Ulterior, am construit o etapă de preprocesare și reprezentare numerică bazată pe tokenizare cu `spaCy`, vocabular cu token-uri speciale (`<PAD>`, `<UNK>`, `<URL>`), embeddings FastText pre-antrenate și padding la lungime fixă. Pentru modelare, am evaluat arhitecturi recurente (RNN și LSTM), explorând hiperparametri precum dimensiunea stării ascunse, numărul de straturi, dropout, direcționalitatea (uni/bidirecțional) și strategii de pooling. Selecția modelelor a fost realizată pe un set de validare folosind F1 și *early stopping*, iar pentru stabilitate am folosit *gradient clipping*. În final, am analizat efectul augmentării simple la nivel de cuvinte (delete/swap/insert), comparând curbele de antrenare și performanța pe test. Rezultatele indică o generalizare mai bună pentru LSTM față de RNN și un impact modest al augmentării, care necesită calibrare atentă pentru a evita introducerea de zgomot semantic.

1 Introducere

Clasificarea sentimentului pe date text este o problemă fundamentală în procesarea limbajului natural, cu aplicații directe în analiza feedback-ului utilizatorilor, monitorizarea opiniei publice și filtrarea automată a conținutului. Spre deosebire de datele tabulare, textul necesită transformarea într-o reprezentare numerică, iar performanța finală depinde atât de calitatea preprocesării, cât și de capacitatea modelului de a surprinde dependențe secvențiale și indicii locale de polaritate.

În această lucrare studiem o problemă de clasificare binară a sentimentului (pozitiv/negativ) în limba română. Provocările principale provin din variabilitatea mare a lungimii textelor, prezența zgomotului (diacritice inconsistene, URL-uri, emoticoane, punctuație neuniformă) și din faptul că indiciile de sentiment pot apărea oriunde în secvență, nu doar la final. Prin urmare, este necesar un pipeline care să reducă variabilitatea superficială a textului, să păstreze informația semantică și să permită antrenarea eficientă a unor modele secvențiale.

2 Analiza setului de date privind sentimentele pe text

2.1 Descrierea setului de date și analiza exploratorie

În această parte folosim setul de date *ro_sent* (recenzii în limba română, etichetate binar: *pozitiv* / *negativ*), disponibil în format CSV (`train.csv`, `test.csv`). Setul conține 17.941 exemple pentru antrenare și 11.005 pentru testare.

2.1.1 Încărcarea datelor și verificări de integritate

Datele au fost descărcate din linkurile oficiale și încărcate în `pandas`. În `train` există o coloană redundantă `index`, iar în `test` apare `Unnamed: 0`; acestea au fost eliminate deoarece nu conțin

informație semantică utilă pentru clasificare.

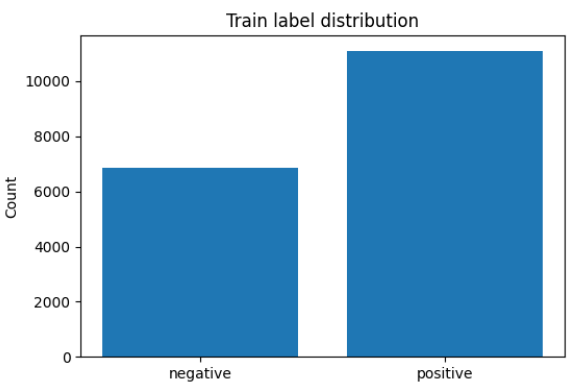
Am verificat valori lipsă și duplicate. în `train` există 290 texte lipsă (`NaN`) și 0 duplicate; în `test` nu există texte lipsă și nu există duplicate. Pentru analizele de text din această secțiune, textele lipsă au fost înlocuite cu șirul vid într-o coloană auxiliară (`text_filled`), fără a modifica textul brut original.

Tabela 1: Descriere dataset

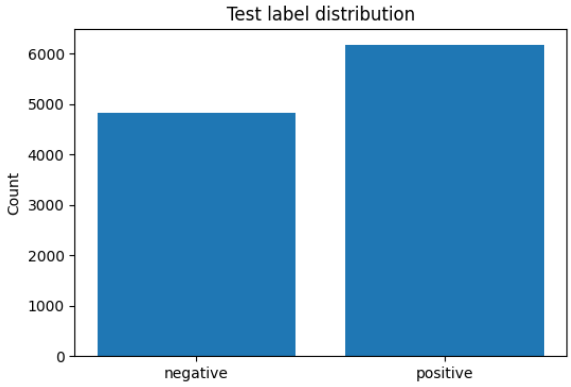
	Train	Test
Număr exemple	17941	11005
Coloane inițiale	(index/text/label)	(Unnamed:0/text/label)
Texte lipsă (NaN)	290	0
Duplicate	0	0

2.1.2 Echilibrul claselor

Conform cerinței, am analizat distribuția etichetelor prin *bar plot* pentru `train` și `test`. În `train`, clasa pozitivă este majoritară (aprox. 61.84%), iar clasa negativă reprezintă aprox. 38.16%. În `test`, distribuția este mai echilibrată (pozitiv 56.13%, negativ 43.87%). Dezechilibrul de clase poate influența atât metrica de acuratețe, cât și pragurile optime de decizie; ulterior, acest aspect poate fi tratat prin ponderi de clasă sau sampling stratificat la împărțirea train/validare.



(a) Distribuția etichetelor - train



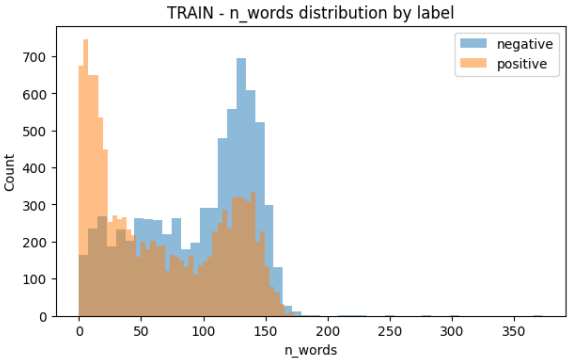
(b) Distribuția etichetelor - test

Figura 1: Comparație între seturile de date

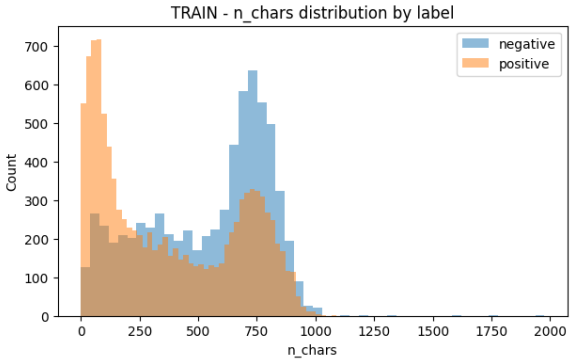
2.1.3 Statistici despre lungimea textelor (cuvinte și caractere)

Am calculat pentru fiecare exemplu: `n_words` (număr cuvinte) și `n_chars` (număr caractere) și am comparat distribuțiile pe etichete (pozitiv/negativ), conform cerinței. Rezultatele indică o diferență relevantă: recenziile negative sunt, în medie, mai lungi decât cele pozitive. De exemplu, media `n_words` este ≈ 95.4 pentru negative și ≈ 65.0 pentru pozitive; similar, `n_chars` are medii ≈ 540 (negativ) vs ≈ 375 (pozitiv). Acest comportament este plauzibil: feedback-ul negativ tinde să includă justificări și detalii suplimentare.

În plus, distribuția prezintă *outlieri*: lungimea maximă observată ajunge la 373 de cuvinte și 1977 caractere (în principal recenzii lungi ce conțin și metadate, ex. user/date). Aceste observații motivează ulterior alegerea unei lungimi fixe de *padding* bazată pe percentilă (ex. percentila 99 pentru `n_words` este 160), astfel încât majoritatea secvențelor să fie păstrate fără cost mare de memorie/compute.



(a) Distribuția `n_words` (train) pe clase.



(b) Distribuția `n_chars` (train) pe clase.

Figura 2: Distribuții ale lungimii textelor pe clase (setul de antrenare).

Tabela 2: Statistici descriptive pentru lungimea textelor (train), pe clase.

Clasă	mean(n_words)	median(n_words)	mean(n_chars)	max(n_words)
negative	95.38	110	540.31	373
positive	64.95	53	374.99	192

2.1.4 Texte lipsă și impact pe clase

Cele 290 texte lipsă din `train` nu sunt distribuite uniform: lipsa apare mult mai des în clasa pozitivă (269 exemple) decât în clasa negativă (21 exemple). Această asimetrie poate introduce un bias dacă exemplele goale sunt tratate ca texte valide; în etapele ulterioare de modelare, aceste instanțe pot fi eliminate sau tratate separat (ex. filtrare înainte de tokenizare) pentru a evita antrenarea pe semnale artificiale.

2.1.5 Semnale de noise și indicii legate de stilul textului

Am analizat câteva tipare simple, potențial predictive, fără a modifica textul: numărul de semne de exclamare (`n_excl`), numărul de semne de întrebare (`n_q`), prezența URL-urilor (`has_url`), prezența cifrelor (`has_num`) și raportul de majuscule (`upper_ratio`).

Rezultatele sugerează diferențe mici, dar interpretabile: `n_q` este mai mare în recenziile negative, ceea ce poate reflecta formulări retorice sau nemulțumiri („de ce...?”). Prezența numerelor este ușor mai frecventă în recenziile negative (ex. referințe la ani, modele, prețuri). URL-urile sunt rare în ambele clase, iar `upper_ratio` este aproape zero (datasetul pare deja normalizat la litere mici în multe cazuri).

Tabela 3: Medii ale semnalelor de „zgomot” (train), pe clase.

Clasă	n_excl	n_q	has_url	has_num	upper_ratio
negative	0.6934	0.3702	0.0028	0.4456	0.0001
positive	0.7240	0.1129	0.0039	0.3559	0.0000

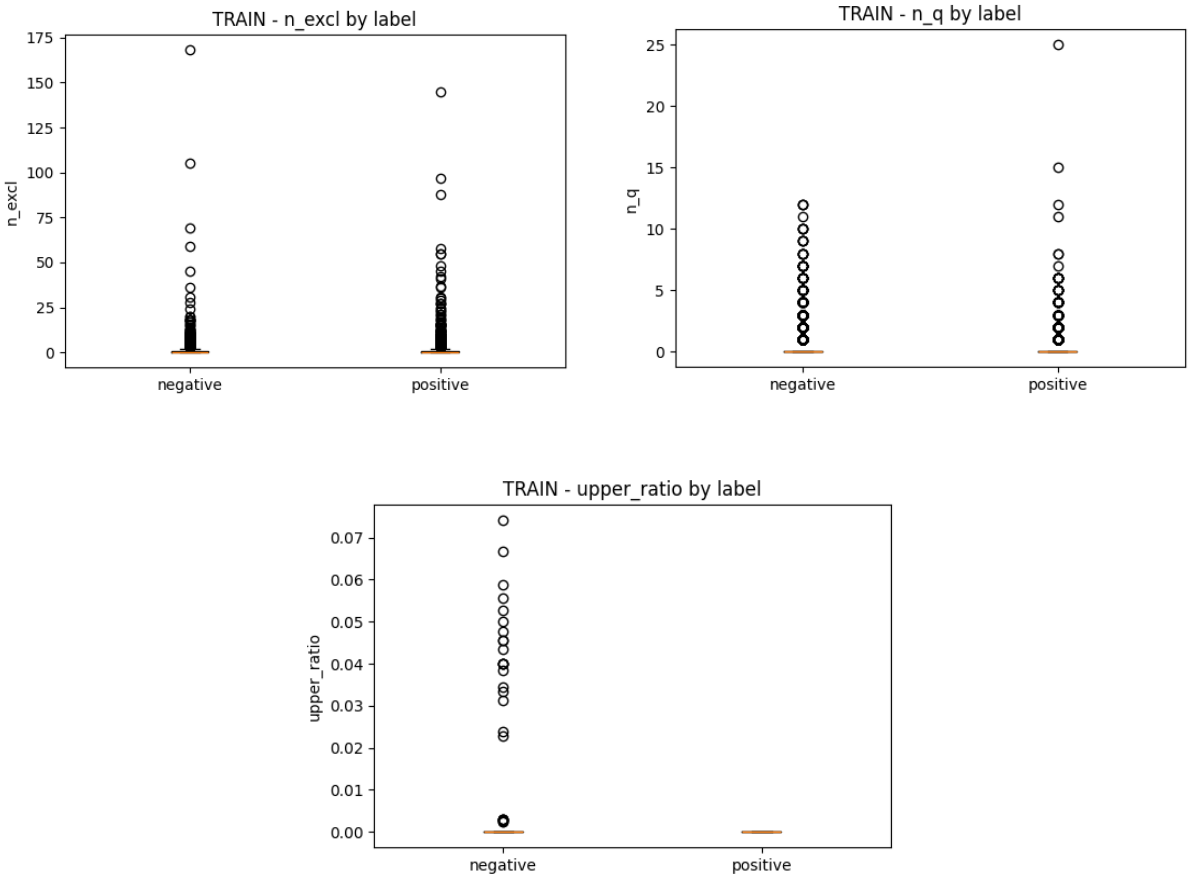


Figura 4: Boxplot-uri pentru semnale de stil (exclamări, întrebări, majuscule) pe clase.

2.1.6 Emoticoane ca indicii de sentiment

Am investigat și emoticoane simple (ex. `:)` / `:-)` / `xd`), deoarece apar în date și pot fi semnale directe de polaritate. Proporția de `has_smiley` și `has_laugh` este semnificativ mai mare în clasa pozitivă, iar `has_sad` apare mai des în clasa negativă. Tokenul `<3` nu apare în setul de antrenare.

Aceste observații motivează păstrarea unor simboluri/emoticoane în preprocesare (sau maparea lor la token-uri speciale), deoarece pot îmbunătăți separabilitatea claselor fără cost suplimentar.

Tabela 4: Rate de apariție pentru emoticoane (train), pe clase.

Clasă	has_smiley	has_sad	has_laugh	has_heart
negative	0.0169	0.0048	0.0022	0.0000
positive	0.0403	0.0027	0.0122	0.0000

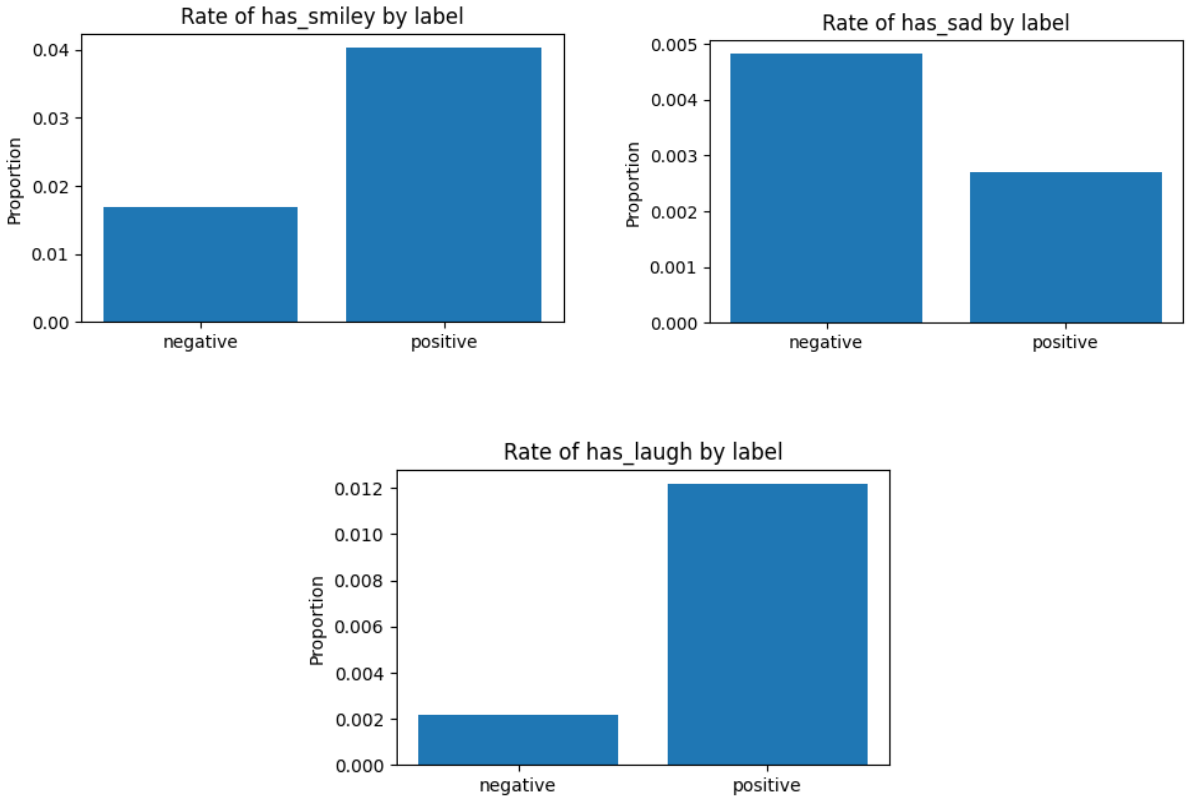


Figura 6: Compararea ratei de apariție a emoticoanelor pe clase.

2.1.7 Cele mai frecvente cuvinte pe clase

Conform cerinței, am identificat cele mai frecvente cuvinte pentru fiecare clasă. Listele sunt dominate de cuvinte care sunt majoritar prepozitii (ex. *de*, *și*, *a*, *în*), însă apar și termeni mai specifici de polaritate: în clasa pozitivă apar mai frecvent *bun*, *bine*, în timp ce în clasa negativă apar termeni precum *rău*. Notabil, negația *nu* este foarte frecventă în ambele clase, motiv pentru care eliminarea stopword-urilor trebuie făcută cu atenție (pentru a nu pierde semnale semantice relevante).

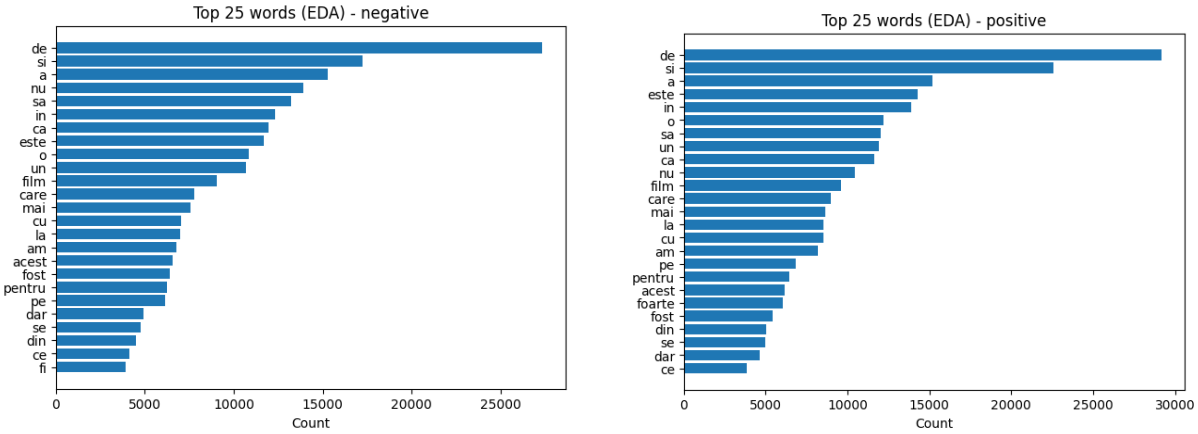


Figura 7: Top cuvinte frecvente în recenzii negative vs. pozitive (train)

2.1.8 Exemple extreme

Am inspectat exemplele cele mai scurte și cele mai lungi. Exemplele cu lungime 0 corespund valorilor lipsă (`NaN` în text). Exemplele foarte lungi conțin adesea metadate (nume utilizator, dată/oră), multiple linii și semne de punctuație repetate, ceea ce poate introduce variabilitate nedorită. Aceste observații susțin aplicarea unei normalizări a spațiilor și filtrarea unor tipare (ex. timestamp) în etapa de preprocesare.

2.1.9 Suprapuneri între train și test

Am verificat suprapuneri exacte de text între `train` și `test`. Au fost găsite 56 exemple identice, toate cu etichete consistente (56/56 potriviri). Proporția este redusă raportat la dimensiunea seturilor și nu schimbă semnificativ concluziile, dar este raportată pentru transparență deoarece poate produce un grad mic de *leakage* (îmbunătățire ușoară a scorului pe test).

2.2 Tokenizare și Embedding Layer

2.2.1 Curățarea și normalizarea textului

Înainte de tokenizare am aplicat o etapă de preprocesare pentru a reduce variațiile inutile din text și pentru a gestiona cazurile speciale. Am eliminat exemplele cu text lipsă sau gol (în setul de antrenare existau 290 astfel de instanțe), pentru a evita introducerea unui semnal artificial în învățare. După filtrare, setul de antrenare rămâne cu 17 651 exemple, iar setul de test cu 11 005 exemple.

Textul a fost normalizat prin:

1. normalizare Unicode (`NFKC`);
2. unificarea diacriticelor românești (ş/ţ → ș/ț);
3. transformare în litere mici;
4. înlocuirea URL-urilor cu un token special `<URL>`;
5. eliminarea caracterelor speciale nepermise (păstrând litere, cifre și punctuație minimă relevantă);
6. normalizarea spațiilor multiple și a liniilor noi.

Această etapă este motivată și de observațiile din EDA, unde apar texte foarte lungi și variabile (inclusiv linii multiple și metadate), iar standardizarea spațiilor reduce variabilitatea fără a pierde conținut semantic important.

2.2.2 Tokenizare (spaCy)

Tokenizarea a fost realizată folosind biblioteca `spaCy` cu modelul românesc `ro_core_news_sm`. Pentru eficiență, componentele costisitoare (tagger, parser, NER, lemmatizer) au fost dezactivate, păstrând doar segmentarea în token-uri. Tokenizarea produce liste de token-uri alfanumerice (inclusiv diacritice), iar tokenul `<URL>` este păstrat explicit.

Eliminarea stopword-urilor este opțională. În această implementare am păstrat stopword-urile (`USE_STOPWORDS=False`), deoarece termeni precum negația (*nu*) pot inversa polaritatea unei recenzii și sunt relevanți pentru clasificarea sentimentului.

2.2.3 Construirea vocabularului și tratarea cuvintelor necunoscute

Vocabularul a fost construit exclusiv pe setul de antrenare, pentru a evita *data leakage*. Pentru fiecare token am calculat frecvența, păstrând token-urile cu frecvență minimă `MIN_FREQ=2` și limitând vocabularul la maximum `MAX_VOCAB=50000` token-uri. În final, vocabularul are 28 882 token-uri.

Au fost introduse token-uri speciale: `<PAD>` (umplere la lungime fixă), `<UNK>` (token necunoscut pentru cuvinte în afara vocabularului), `<URL>` (semnal pentru prezența linkurilor). Token-urile sunt mapate la indici printr-un dicționar `stoi`, iar textele sunt transformate în secvențe numerice.

2.2.4 Padding și reprezentare secvențială

Secvențele numerice au fost normalizate la o lungime fixă `MAX_LEN=160`. Această valoare a fost aleasă pe baza analizei exploratorii (percentila 99 a lungimii în cuvinte este 160), astfel încât majoritatea textelor să fie păstrate aproape integral. Secvențele mai lungi sunt trunchiate, iar cele mai scurte sunt completate cu `<PAD>`. După padding, tensorii au dimensiunile $[17651, 160]$ pentru train și $[11005, 160]$ pentru test.

2.2.5 Embedding preantrenat (fastText)

Pentru reprezentare semantică densă am folosit vectori fastText preantrenați pentru limba română (`cc.ro.300`). Am construit o matrice de embedding de dimensiune $|V| \times 300$, unde pentru token-urile găsite în fastText am încărcat vectorul corespunzător, iar pentru token-urile lipsă am inițializat aleator (cu păstrarea vectorului nul pentru `<PAD>`). Acoperirea obținută este 87.95% din vocabular.

Embedding-ul a fost integrat în PyTorch prin `nn.Embedding.from_pretrained(...)` cu `freeze=False`, permițând ajustarea vectorilor în timpul antrenării: intrarea de formă $[B, 160]$ este mapată într-o reprezentare $[B, 160, 300]$. Astfel, spaCy este utilizat pentru tokenizarea textului în limba română, producând secvențe de token-uri. fastText este utilizat ulterior pentru a inițializa embedding-ul: fiecare token mapat la un indice este transformat într-un vector dens de dimensiune 300, care encodează similarități semantice între cuvinte.

Tabela 5: Sumar preprocesare.

Parametru / element	Valoare
Exemple train după filtrare	17651
Exemple test	11005
MAX_LEN (padding)	160
Dimensiune embedding	300
Vocab size	28882
Acoperire fastText	87.95%
Token-uri speciale	<PAD>, <UNK>, <URL>

2.3 Arhitectură RNN simplă

2.3.1 Pregătirea datelor pentru RNN: split train/validare și batch-uri

După etapa de tokenizare, mapare indici și padding la lungime fixă, am obținut un tensor `train_padded` de dimensiune $[N, T]$, unde $T = 160$ este lungimea maximă aleasă (pe baza percentilelor din EDA), iar `y_train` conține etichetele binare (0 = negativ, 1 = pozitiv). Pentru selecția hiperparametrilor și prevenirea overfitului, am împărțit setul de antrenare într-un *train split* (85%) și un *validation split* (15%), folosind stratificare după etichetă pentru a păstra

distribuția claselor. Datele sunt încărcate în batch-uri (`batch size` = 128) folosind `DataLoader`, rezultând 118 batch-uri pentru train și 21 pentru validare.

Întrucât secvențele sunt pad-uite, am calculat și lungimea reală a fiecărui exemplu ca numărul de tokeni diferiți de `<PAD>`. Acest lucru permite utilizarea `pack_padded_sequence`, astfel încât rețeaua recurentă să proceseze doar tokenii reali, evitând efectele nedorite ale padding-ului asupra dinamicii stărilor ascunse și reducând costul de calcul.

2.3.2 Modelul RNN și motivația arhitecturii

Modelul de tip RNN simplu folosește următoarea structură:

- **Embedding** inițializat cu vectori fastText preantrenați (dimensiune 300) și *fine-tuned*;
- **nn.RNN** cu activare `tanh`, cu hiperparametri variați: dimensiunea stării ascunse (`hidden_size`) și numărul de straturi (`num_layers`);
- **Dropout** pentru regularizare;
- **Strat liniar** (`Linear`) care proiectează reprezentarea finală în 2 logit-uri (negativ/pozitiv).

Pentru clasificare, reprezentarea textului este obținută din ultimul hidden state al ultimului strat h_T care agregheaza informația secvențială acumulată în timpul parcurgerii tokenilor, fiind o alegere standard pentru un baseline recurent simplu. Dropout-ul este aplicat înaintea stratului final pentru a reduce overfitul.

2.3.3 Strategia de antrenare: optimizare, dezechilibru de clase, stabilitate

Problema are distribuție neuniformă a claselor (pozitivul este mai frecvent decât negativul), astfel că am folosit `CrossEntropyLoss` cu ponderi de clasă inverse proporționale cu frecvențele din train (echivalent cu a penaliza mai mult erorile pe clasa rară). Acest lucru este important mai ales când raportăm performanța prin F1, care este sensibil la compromisurile precizie/recall.

Optimizarea se face cu `AdamW` (lr = 10^{-3}, weight_decay = 10^{-4}), ales pentru stabilitate și regularizare L2 implicită prin decay. În plus, pentru rețele recurente există riscul de *exploding gradients*, motiv pentru care am aplicat **gradient clipping** la 1.0. Antrenarea rulează cu **early stopping** (patience = 6) monitorizând **F1 pe validare**: dacă F1 nu se îmbunătățește timp de 6 epoci consecutive, antrenarea se oprește și se revine la cel mai bun checkpoint (după F1).

2.3.4 Selecția hiperparametrilor și interpretarea comportamentului pe validare

Am explorat un set restrâns de configurații (*grid search* manual) variind:

- `hidden_size` $\in \{128, 256\}$: capacitatea stării ascunse (câtă informație poate stoca RNN-ul);
- `num_layers` $\in \{1, 2\}$: profunzimea modelului (mai multe straturi pot modela interacțiuni mai complexe, dar cresc riscul de overfitting și instabilitate);
- `dropout` $\in \{0.2, 0.3, 0.4\}$: regularizare (probabilitatea de a opri neuroni în timpul antrenării).

Rezultatele pe validare sunt sumarizate în Tabelul 6, unde raportăm pentru fiecare configurație: epoca optimă (după F1), numărul de epoci rulate până la early stopping, F1/accuracy/loss la epoca optimă.

Tabela 6: Rezultate pe validare pentru configurațiile RNN (selecție după best `val_f1`).

Config	hidden	layers	dropout	best ep.	best val F1	stop
rnn_h128_l2_d0.3	128	2	0.3	4	0.8767	early stop
rnn_h128_l1_d0.2	128	1	0.2	12	0.8729	early stop
rnn_h256_l2_d0.3	256	2	0.3	2	0.8726	early stop
rnn_h256_l1_d0.4	256	1	0.4	4	0.8721	early stop
rnn_h256_l1_d0.2	256	1	0.2	2	0.8709	early stop
rnn_h128_l1_d0.4	128	1	0.4	5	0.8622	early stop

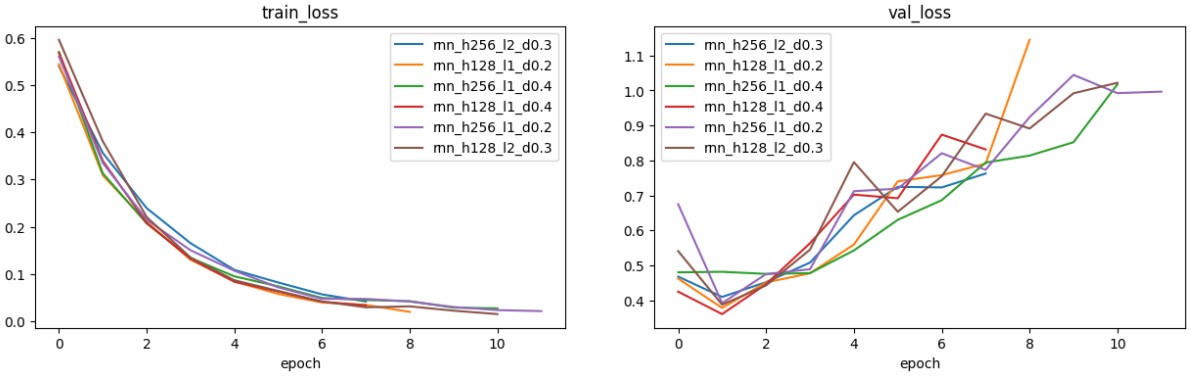
Din tabel se observă că cea mai bună configurație (după F1 pe validare) a fost `rnn_h128_l2_d0.3`. Intuitiv, 2 straturi oferă capacitate suplimentară față de 1 strat, dar un `hidden_size` prea mare (256) nu a adus câștig consistent, sugerând că pentru acest dataset RNN-ul simplu atinge rapid limita de expresivitate și începe să supraînvețe. În același timp, dropout prea mare (0.4) a degradat performanța pentru hidden=128 (modelul devine prea regularizat și pierde semnal util), iar dropout prea mic (0.2) poate permite overfitting mai repede (val_loss crește mai agresiv).

Curbele de antrenare (Figura **??**) arată tiparul clasic pentru un model cu risc de overfitting: **train_loss scade monoton**, în timp ce **val_loss** poate crește după un număr mic de epoci. Acest lucru indică faptul că modelul devine din ce în ce mai „încrezător" pe exemplele din train, dar generalizează mai slab pe validare. Mai mult, curbele de **val_f1 sunt relativ "spiky"** (variază de la o epocă la alta). Acest comportament este normal deoarece:

(i) setul de validare este relativ mic (15%);

(ii) schimbări mici în decizie pot modifica F1 (mai ales când există dezechilibru de clase);

(iii) optimizarea stocastică produce actualizări ce pot îmbunătăți temporar separarea claselor fără a îmbunătăți consistent probabilitățile (ceea ce se reflectă în loss).



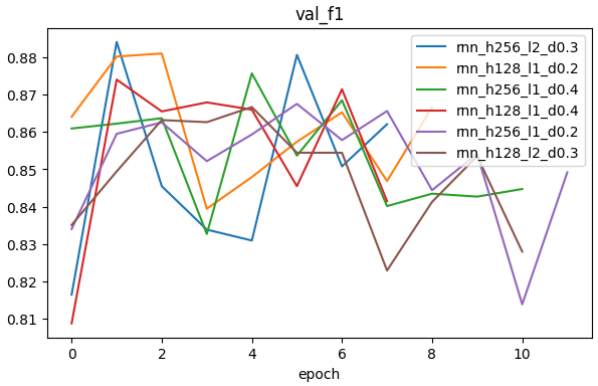


Figura 9: Curbe de antrenare/ validare pentru configurațiile RNN: `train_loss`, `val_loss` și `val_f1`.

2.3.5 Evaluarea configurației selectate pe test și interpretarea metricilor

După selecția pe validare, am evaluat modelul `rnn_h128_l2_d0.3` pe setul de test:

$$\text{loss} = 0.6886, \quad \text{acc} = 0.8047, \quad \text{F1} = 0.8365.$$

Pentru o interpretare mai fină a performanței, am raportat:

- **F1 (binary)**: în `sklearn` această variantă consideră implicit clasa 1 ca pozitivă și calculează F1 pentru această clasă. Practic, măsoară cât de bine identificăm **pozitivul**.
- **F1 (macro)**: media aritmetică a F1 pentru fiecare clasă (negativ și pozitiv). Este mai echitabilă când există dezechilibru, deoarece tratează ambele clase cu aceeași importanță.

Am inclus și `classification_report` (precision/recall/F1 pe fiecare clasă) și matricea de confuzie. Aceasta evidențiază tipul de erori: câte negative sunt prezise ca pozitive și invers. Un aspect relevant în astfel de date este că un model poate obține accuracy bun, dar să sacrifice recall-ul clasei rare; de aceea raportarea F1 macro și analiza matricei de confuzie sunt necesare.

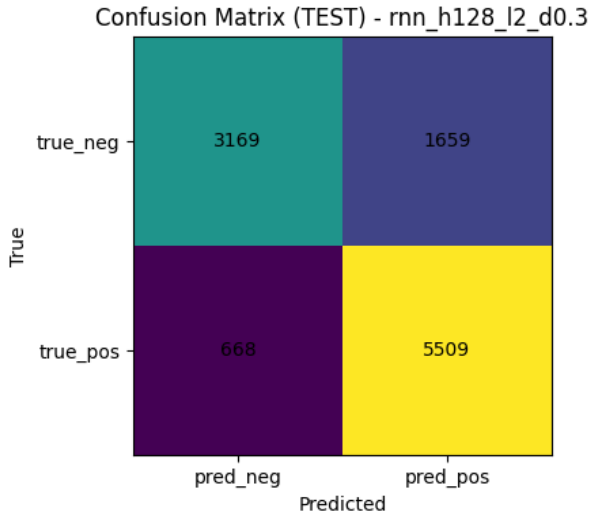


Figura 10: Matricea de confuzie pentru modelul RNN selectat (`rnn_h128_l2_d0.3`) pe test.

Tabela 7: Classification report (TEST)

Class	Precision	Recall	F1-score	Support
negative	0.83	0.66	0.73	4828
positive	0.77	0.89	0.83	6177
Accuracy		0.79		11005
Macro avg	0.80	0.77	0.78	11005
Weighted avg	0.79	0.79	0.78	11005

2.3.6 Augmentare pentru text: Random Delete/Swap/Insert și comparație cu baseline

Pentru îmbunătățirea generalizării, am experimentat augmentări simple la nivel de tokeni, inspirate din tehnici standard:

- **Random Delete**: ștergere aleatorie a tokenilor cu probabilitate $p = 0.1$;
- **Random Swap**: interschimbarea aleatorie a două poziții (1 swap);
- **Random Insert**: inserarea unui token aleator (1 inserție) ales din vocabular (fără tokenii speciali).

Am generat augmentări pentru 30% din exemplele din *train split*, adăugând câte o copie augmentată pentru acele exemple. Astfel, train split-ul a crescut de la 15003 la 19503 secvențe, în timp ce setul de validare a rămas neschimbat (pentru o comparație corectă).

Figura de mai jos compară evoluția pe validare (F1 și loss) pentru modelul selectat cu și fără augmentare. Observăm că augmentarea poate modifica dinamica antrenării: uneori F1 crește temporar (mai ales dacă augmentarea introduce variații utile), dar loss-ul poate crește deoarece exemplele augmentate sunt mai „zgomotoase" și modelul devine mai puțin încrezător. Acest fenomen este normal: augmentarea urmărește să reducă overfitting-ul și să crească robustetea, nu neapărat să minimizeze imediat loss-ul.

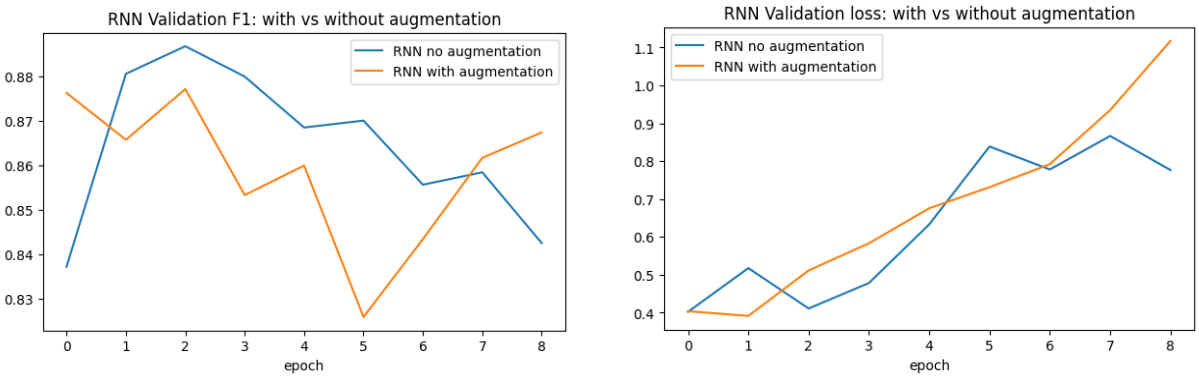


Figura 11: Comparattie RNN cu si fără augmentare: evoluttia valf1 si valloss

Pe test, rezultatele au fost:

RNN no-aug: acc = 0.8075, F1 = 0.8243 RNN with-aug: acc = 0.8003, F1 = 0.8296.

Deși accuracy a scăzut ușor cu augmentare, F1 (binary) a crescut, indicând o posibilă îmbunătățire a identificării clasei pozitive. Totuși, F1 macro a scăzut (0.8058 → 0.7942), ceea ce sugerează un compromis: augmentarea a mutat frontiera de decizie astfel încât modelul a devenit mai bun la pozitiv (recall mai mare), dar mai slab la negativ. Această interpretare este confirmată de rapoartele pe clasă și de matricile de confuzie (Figura **??**), unde varianta cu augmentare are

mai multe confuzii pentru *true_neg* (negativ prezis ca pozitiv), dar mai puține confuzii pentru *true_pos*.

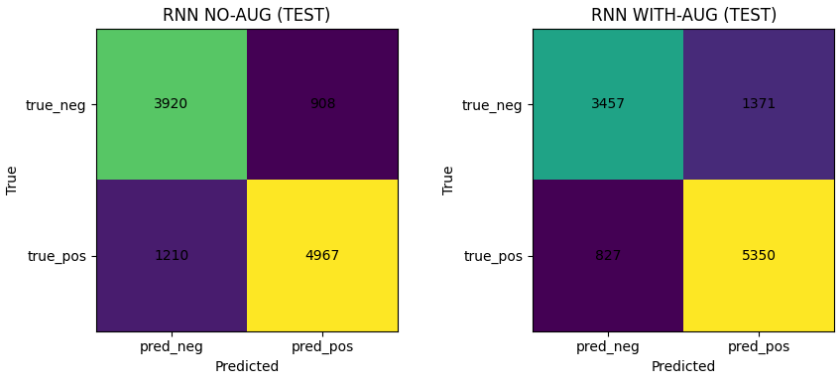


Figura 12: Matricea de confuzie (și varianta normalizată pe rând) pentru RNN fără augmentare pe test.

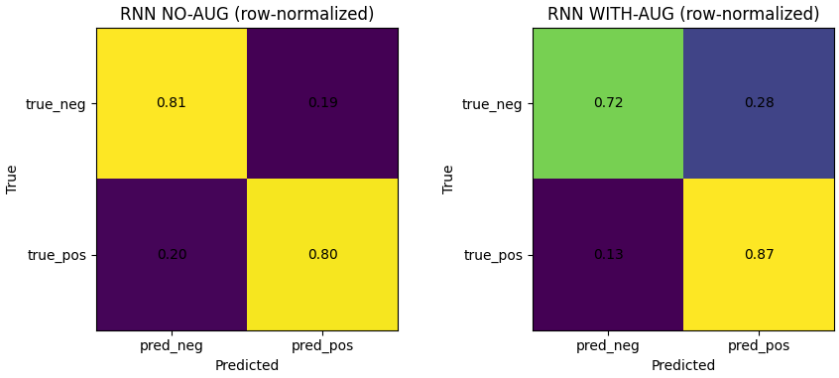


Figura 13: Matricea de confuzie (și varianta normalizată pe rând) pentru RNN cu augmentare, pe test.

Tabela 8: Classification report (TEST) – RNN NO-AUG

Class	**Precision**	**Recall**	**F1-score**	**Support**
negative	0.76	0.81	0.79	4828
positive	0.85	0.80	0.82	6177
Accuracy			0.81	11005
Macro avg	0.80	0.81	0.81	11005
Weighted avg	0.81	0.81	0.81	11005

Tabela 9: Classification report (TEST) – RNN WITH-AUG

Class	**Precision**	**Recall**	**F1-score**	**Support**
negative	0.81	0.72	0.76	4828
positive	0.80	0.87	0.83	6177
Accuracy			0.80	11005
Macro avg	0.80	0.79	0.79	11005
Weighted avg	0.80	0.80	0.80	11005

În concluzie, RNN-ul simplu oferă un baseline rezonabil, însă prezintă tendințe de overfitting (val_loss crește după câteva epoci), iar augmentarea produce îmbunătățiri modeste și un compromis între clase. Aceste observații motivează explorarea unui model recurent mai expresiv (LSTM), care poate modela mai bine dependențe pe termen lung și poate fi mai robust la variații ale textului.

2.4 Arhitectură LSTM

Spre deosebire de un RNN simplu (cu funcție de activare *tanh*), o rețea de tip LSTM (*Long Short-Term Memory*) introduce mecanisme de tip *gates* (input/forget/output) care controlează ce informație se păstrează sau se uită. Acest lucru permite modelului să capteze mai bine dependențe pe secvențe mai lungi, relevante în analiza de sentiment (de ex. negații, contraste de tipul "bun, dar...").

Am folosit aceeași împărțire `train/val` ca la RNN (85% / 15%, stratificat pe etichete), cu `batch_size=128`. Intrările sunt secvențe **padded/truncate** la `MAX_LEN=160` token-uri (aleasă din EDA pe baza percentilelor), iar embedding-urile inițiale sunt fastText RO (300 dim), lăsate **antrenabile** (`freeze=False`). Optimizarea s-a făcut cu **AdamW** (`lr=1e-3`, `weight_decay=1e-4`). Pentru stabilitate la antrenare am aplicat **gradient clipping** `clip=1.0`. Din cauza dezechilibrului de clase, am folosit **ponderi inverse ale frecvenței** în `CrossEntropyLoss`.

2.4.1 Arhitectura modelului.

Modelul LSTM implementat are următoarea structură:

- **Embedding layer**: transformă indicii token-urilor în vectori densi (300 dim).
- **LSTM encoder**: primește secvența de embedding-uri și produce stări ascunse. Am folosit `pack_padded_sequence` pentru a ignora padding-ul în calcul.
- **Pooling / reprezentare de propoziție** (`pooling`):
  - `last`: folosește ultimul hidden state (sau concatenarea forward/backward în cazul bidirecțional).
  - `mean`: face media stărilor pe timp (cu mască pentru padding), captând mai uniform semnalul pe toată propoziția.
- **Head de clasificare**: un `MLP` scurt (`Linear-ReLU-Dropout-Linear`) peste reprezentarea finală.

2.4.2 Hiperparametri explorați și semnificația lor.

Am evaluat mai multe configurații variind:

- **hidden_size** (128 vs 256): capacitatea internă a LSTM (mai mare => mai expresiv, dar risc mai mare de overfitting).
- **bidirectional** (False/True): în varianta bidirecțională, modelul vede contextul din ambele sensuri, util pentru texte unde cuvintele de la final pot schimba interpretarea celor de la început.
- **pooling** (`last` vs `mean`): `mean` tinde să fie mai robust la poziția exactă a cuvintelor relevante, în timp ce `last` se bazează mai mult pe capacitatea LSTM de a sintetiza informația în ultima stare.
- **dropout** (fix 0.3 aici): regularizare pentru a reduce overfitting-ul.

Am păstrat `num_layers=1` pentru toate configurațiile LSTM din acest experiment, deoarece setul este relativ mare, iar creșterea adâncimii LSTM crește parametrii și instabilitatea; în plus, primele rezultate au indicat că overfitting-ul apare rapid (după câteva epoci), deci adâncimea suplimentară nu era justificată.

2.4.3 Selecția modelului pe validare.

Pentru fiecare configurație am monitorizat **val_F1** și am aplicat **early stopping** cu `patience=6` (pe criteriul **val_F1**, nu pe loss). Motivația este că obiectivul problemei este performanța de clasificare (în special pe clasa pozitivă), iar loss-ul (cross-entropy) poate continua să crească atunci când modelul devine prea încrezător pe exemplele greșite. Curbele de `val_loss` și `val_F1` pot părea „spiky" deoarece:

- setul de validare este relativ mic (15%),
- mici variații ale frontierei de decizie pot schimba F1 (mai ales în prezența dezechilibrului),
- optimizarea stocastică poate îmbunătăți temporar separarea claselor fără a calibra bine probabilitățile (afectând loss-ul).

2.4.4 Rezultate pe validare (fără augmentare).

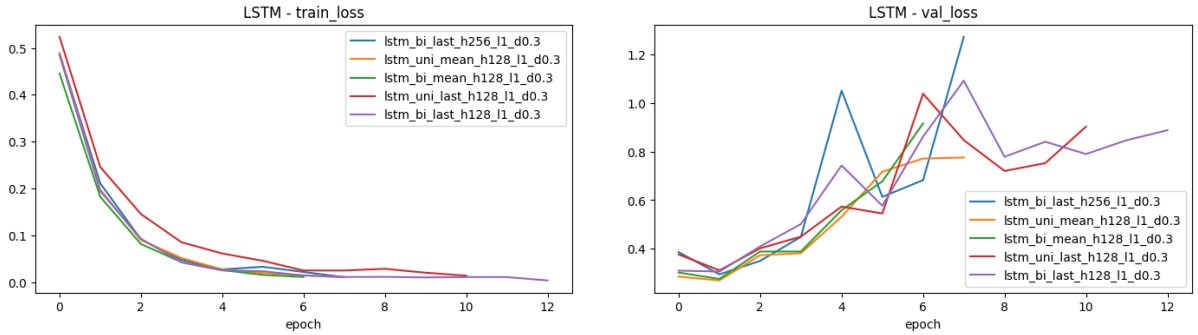
Tabelul 10 rezumă performanța maximă pe validare pentru fiecare configurație. În mod interesant, cea mai bună valoare a fost obținută de o configurație **bidirecțională cu hidden mai mare** (`h=256`) și pooling `last`. Intuitiv, capacitatea crescută și contextul bidirecțional ajută la captarea dependențelor lungi și a nuanțelor de sentiment.

Tabela 10: LSTM: selecția configurațiilor după `best_val_F1`.

Name	hidden	bidir	pool	best_val_F1	best_ep	ep_ran
lstm_bi_last_h256_l1_d0.3	256	True	last	0.9127	2	8
lstm_uni_mean_h128_l1_d0.3	128	False	mean	0.9104	2	8
lstm_bi_mean_h128_l1_d0.3	128	True	mean	0.9104	1	7
lstm_uni_last_h128_l1_d0.3	128	False	last	0.9017	5	11
lstm_bi_last_h128_l1_d0.3	128	True	last	0.9000	7	13

2.4.5 Graficele de antrenare.

În figurile de mai jos se observă tiparul clasic: `train_loss` scade monoton (modelul învață să potrivească datele de train), în timp ce `val_loss` tinde să crească după 1–3 epoci, semn de **overfitting**. Totuși, `val_F1` atinge un maxim timpuriu și apoi oscilează: asta indică faptul că modelul rearanjează frontiera de decizie într-un mod care nu mai generalizează consistent. De aceea, **early stopping** este esențial: păstrăm checkpoint-ul de la epoca cu **cel mai bun** `val_F1`.



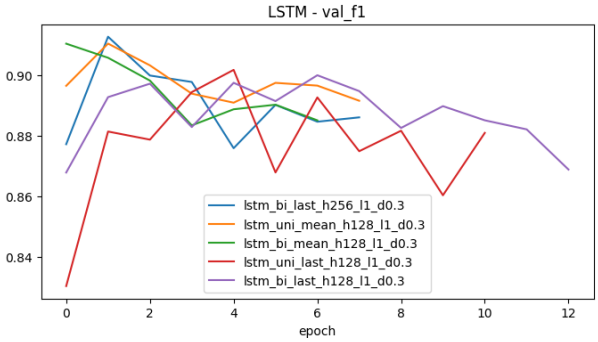


Figura 15: Grafice LSTM train + val

Conform selecției pe validare, am evaluat pe test configurația `lstm_bi_last_h256_l1_d0.3`, obținând:

$$\texttt{loss} = 0.3538, \quad \texttt{acc} = 0.8509, \quad \texttt{F1(binary)} = 0.8628.$$

Matricea de confuzie (Figura *LSTM best, TEST*) arată: TN=4205, FP=623, FN=1018, TP=5159. Modelul recunoaște bine clasa pozitivă (precision mare), dar are un număr non-neglijabil de false negative, ceea ce sugerează existența unor exemple pozitive ambigue sau cu vocabular rar.

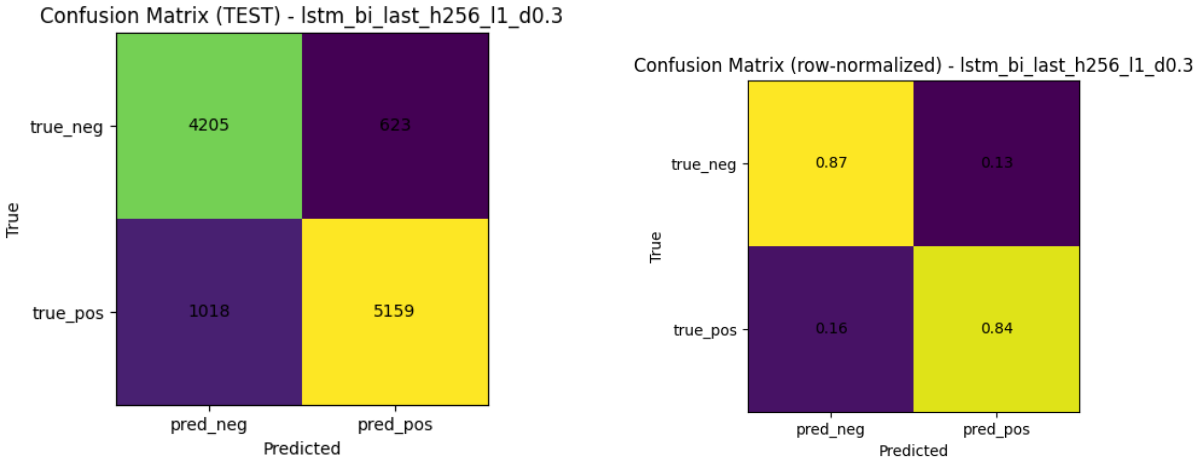


Figura 16: Matricile de confuzie dupa evaluarea pe test

`F1(binary)` (implicit în sklearn) se referă la F1 pentru **clasa pozitivă** (label=1). `F1(macro)` este media aritmetică a F1 pe **fiecare clasă**, tratând clasele egal, deci penalizează mai mult performanța slabă pe clasa minoritară. În contexte cu dezechilibru, macro-F1 este un indicator mai robust.

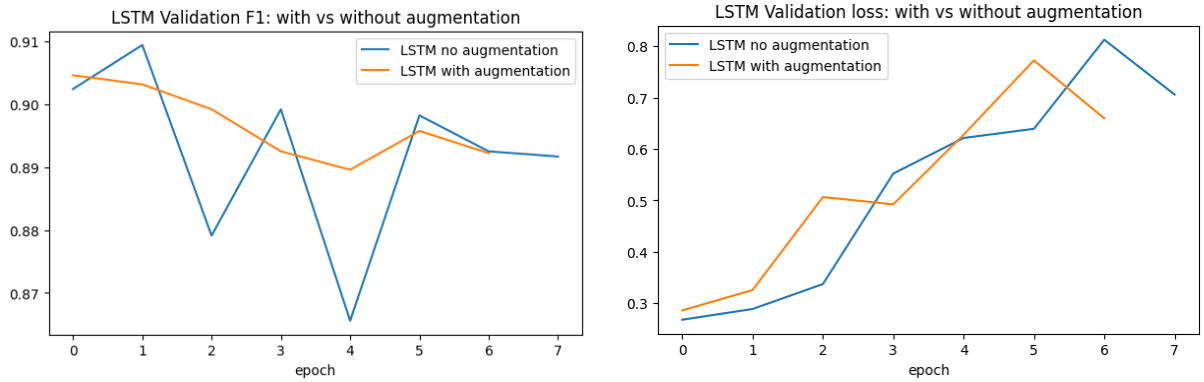
Tabela 11: Classification report (TEST) – LSTM `lstm_bi_last_h256_l1_d0.3`.

Class	Precision	Recall	F1-score	Support
negative	0.81	0.87	0.84	4828
positive	0.89	0.84	0.86	6177
Accuracy		0.85		11005
Macro avg	0.85	0.85	0.85	11005
Weighted avg	0.85	0.85	0.85	11005

2.4.6 Augmentare pentru LSTM (comparativ cu fără augmentare).

Pentru a evalua efectul augmentării, am folosit aceleași operații simple ca la RNN: *Random Delete* (prob. 0.1), *Random Swap* (1 swap) și *Random Insert* (1 insert), aplicate pe un subset de 30% din *train* (fiecare probă aleasă primește o copie augmentată). Am ales pentru această comparație configurația `lstm_bi_mean_h128_l1_d0.3` (performanță foarte apropiată de vârf pe validare, dar cu dimensiune mai mică, deci mai stabilă și mai ieftină computațional) și am rulat două antrenări: *no-aug* vs. *with-aug*, cu aceeași schemă de *early stopping*.

Înainte de evaluarea finală pe test, am analizat curbele pe **setul de validare** (Figura 17). Se observă că varianta *no-aug* are oscilații mai mari ale `val_F1` (variantă mai ridicată a generalizării), în timp ce *with-aug* produce o curbă mai „netedă", sugerând un efect de regularizare: modelul devine mai puțin sensibil la particularitățile datelor de antrenare. În paralel, `val_loss` tinde să crească după câteva epoci pentru ambele setări, indicând început de *overfitting* (modelul devine tot mai încrezător pe predicții greșite, iar cross-entropy penalizează puternic aceste cazuri). Prin urmare, selecția modelului final s-a făcut pe baza celui mai bun `val_F1` (early stopping), apoi am raportat performanța pe test.



(a) `val_F1`: no-aug vs. with-aug

(b) `val_loss`: no-aug vs. with-aug

Figura 17: Curbe de validare pentru LSTM: efectul augmentării.

Pe **setul de test**, diferențele au fost mici:

$$\text{NO-AUG: } \text{loss} = 0.3455,\ \text{acc} = 0.8539,\ \text{F1} = 0.8699$$

$$\text{WITH-AUG: } \text{loss} = 0.3366,\ \text{acc} = 0.8529,\ \text{F1} = 0.8678$$

Augmentarea nu a adus un câștig consistent de F1 pe test în acest caz, dar a contribuit la stabilizarea curbei pe validare, ceea ce sugerează că efectul principal a fost reducerea sensibilității la variațiile din date, nu neapărat îmbunătățirea separării finale a claselor.

LSTM are deja capacitate suficientă și generalizează bine; augmentarea simplă (swap/delete/insert) poate introduce zgomot lexical care nu ajută semnalul de sentiment (sau chiar degradează ușor). Totuși, faptul că scorurile rămân apropiate sugerează că modelul este relativ robust la perturbări moderate.

Tabela 12: LSTM: comparație pe TEST (no-aug vs with-aug) pentru `lstm_bi_mean_h128_l1_d0.3`.

Variantă	Test loss	Test acc	Test F1(binary)
NO-AUG	0.3455	0.8539	0.8699
WITH-AUG	0.3366	0.8529	0.8678

Tabela 13: Classification report (TEST) – LSTM NO-AUG.

Class	Precision	Recall	F1-score	Support
negative	0.83	0.83	0.83	4828
positive	0.87	0.87	0.87	6177
Accuracy		0.85		11005
Macro avg	0.85	0.85	0.85	11005
Weighted avg	0.85	0.85	0.85	11005

Tabela 14: Classification report (TEST) – LSTM WITH-AUG.

Class	Precision	Recall	F1-score	Support
negative	0.83	0.84	0.83	4828
positive	0.88	0.86	0.87	6177
Accuracy		0.85		11005
Macro avg	0.85	0.85	0.85	11005
Weighted avg	0.85	0.85	0.85	11005

2.4.7 Interpretarea matricilor de confuzie (no-aug vs aug).

Pentru NO-AUG, matricea de confuzie indică (TN=4023, FP=805, FN=803, TP=5374), iar pentru WITH-AUG (TN=4071, FP=757, FN=862, TP=5315). Augmentarea a redus ușor FP (mai puține negative confundate ca pozitive), dar a crescut FN (mai multe pozitive ratate), ceea ce explică scăderea mică a `F1(binary)`. Per ansamblu, diferențele sunt mici și nu indică un câștig clar pentru această schemă simplă de augmentare.

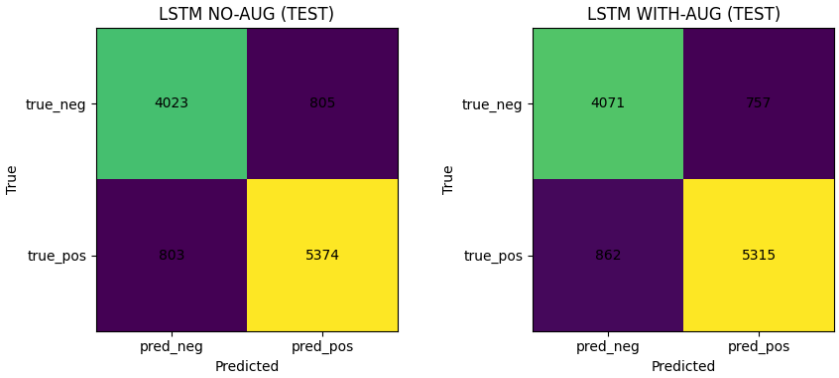


Figura 18: Matrice de confuzie

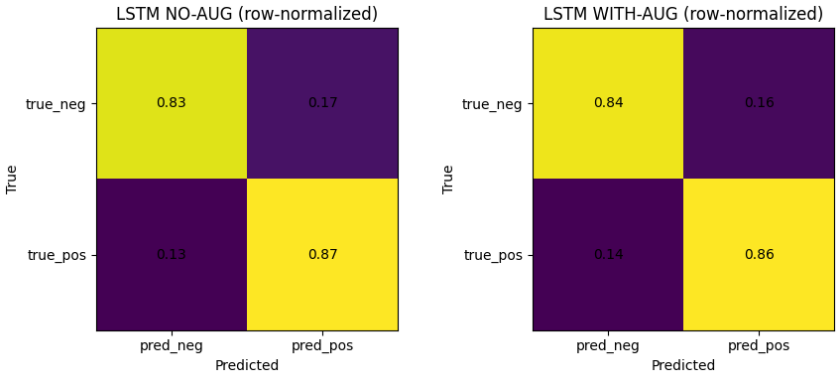


Figura 19: Matrice de confuzie normalizata

3 Concluzii

În cadrul acestei lucrări am urmărit un flux complet de învățare automată pe date text: analiză exploratorie, preprocesare, reprezentare numerică și antrenarea de modele recurente. EDA a arătat că textele au lungimi variabile și conțin zgomot (punctuație repetată, emoticoane, URL-uri, metadate), ceea ce justifică normalizarea și alegerea unei lungimi fixe pentru secvențe. Reprezentarea prin tokenizare + embeddings pre-antrenate (FastText) a oferit un punct de plecare solid, reducând nevoia de a învăța de la zero semnificația cuvintelor.

La nivel de modelare, am observat comportamentul clasic de *overfitting*: loss-ul pe train scade constant, în timp ce pe validare poate crește după câteva epoci, deși F1 poate rămâne bun. Acest lucru indică faptul că modelul ajunge să se potrivească prea bine pe train și că selecția checkpoint-ului trebuie făcută pe validare (early stopping), nu după numărul maxim de epoci. Comparativ, LSTM-ul a fost mai robust decât RNN-ul simplu, ceea ce confirmă că mecanismele de gating ajută la păstrarea informației relevante pe secvențe mai lungi. În plus, strategiile de agregare a informației (pooling peste timp) au fost competitive, sugerând că decizia de sentiment depinde de distribuția indiciilor pe tot textul, nu doar de finalul secvenței.

Augmentările simple la nivel de cuvinte (delete/swap/insert) au produs diferențe mici pe test: uneori îmbunătățesc ușor F1, alteori îl degradează marginal. Concluzia practică este că, în această problemă, performanța este dominată de calitatea reprezentării (embedding-uri + tokenizare) și de controlul overfitting-ului (regularizare, early stopping), iar augmentarea trebuie calibrată atent pentru a nu introduce zgomot care schimbă sensul propozițiilor. Per total, abordarea cu LSTM + embeddings pre-antrenate oferă un compromis bun între performanță și complexitate, iar evaluarea pe set de test confirmă că modelul generalizează rezonabil, cu erori explicabile prin ambiguitate semantică și prezența zgomotului în date.