



UNIVERSITATEA DIN  
BUCUREȘTI

FACULTATEA DE  
MATEMATICĂ ȘI  
INFORMATICĂ



SPECIALIZAREA INFORMATICĂ

Lucrare de licență

APLICAȚIE WEB PENTRU VIDEOCLIP-URI CU  
FUNCȚII DE ÎNȚELEGEREA VORBIRII ȘI  
REGĂSIRE PE BAZĂ DE TEXT

Absolvent

Trifan Robert-Gabriel

Coordonator științific

Prof. Dr. Ionescu Radu

București, iunie 2024

## **Rezumat**

Popularitatea videoclipurilor a avut o creștere constantă în ultimii ani, reprezentând 82.5% din traficul web în 2023. Statistici recente arată că oamenii petrec în media 17 ore pe săptămâna vizionând videoclipuri online, acestea fiind cu 52% mai predispuse să fie distribuite pe rețelele de socializare decât alte tipuri de conținut. [3]

În acest context, lucrarea de față își propune să contribuie la creșterea accesibilității și personalizării conținutului video, prin dezvoltarea unei aplicații web care permite adăugarea de subtitrări, căutarea videoclipurilor pe bază de text și clasificarea acestora în funcție de conținutul lor. Aplicația oferă utilizatorilor posibilitatea de a vizualiza subtitrările în timp real și de a căuta cuvinte cheie în metadatele videoclipurilor precum titlu, descriere, topic și subtitrare.

## **Abstract**

The popularity of videos has been steadily increasing in recent years, representing 82.5% of web traffic in 2023. Recent statistics show that people spend an average of 17 hours a week watching online videos, which are 52% more likely to be shared on social networks than other types of content. [3]

In this context, this work aims to contribute to increasing the accessibility and personalization of video content, by developing a web application that allows the addition of subtitles, searching for videos based on text and classifying them according to their content. The application offers users the possibility to view subtitles in real time and to search for keywords in the metadata of videos such as title, description, topic and subtitle.

# Cuprins

<b>1</b>	<b>Introducere</b>	<b>5</b>
1.1	Motivație . . . . .	5
1.2	Domenii abordate . . . . .	5
1.3	Structura lucrării . . . . .	6
<b>2</b>	<b>Inteligență artificială</b>	<b>7</b>
2.1	Recunoașterea vorbirii . . . . .	7
2.1.1	Arhitectura modelului . . . . .	7
2.1.2	Setul de date . . . . .	10
2.1.3	Antrenarea modelului . . . . .	11
2.1.4	Îmbunătățire cu n-gram language model . . . . .	13
2.2	Clasificarea videoclipurilor . . . . .	14
2.2.1	Arhitectura modelului . . . . .	14
2.2.2	Setul de date . . . . .	15
2.2.3	Antrenarea modelului . . . . .	16
2.3	Concluzie . . . . .	17
<b>3</b>	<b>Inginerie software</b>	<b>19</b>
3.1	Arhitectura aplicației . . . . .	19
3.2	Frontend . . . . .	20
3.2.1	React . . . . .	20
3.2.2	Autentificare . . . . .	21
3.2.3	Video . . . . .	21
3.2.4	Căutare . . . . .	24
3.2.5	Temă . . . . .	24
3.2.6	Rute protejate . . . . .	25
3.3	Backend . . . . .	26
3.3.1	Express Server - Procesarea cererilor . . . . .	26
3.3.2	Flask Server - Recunoașterea vorbirii . . . . .	33
3.3.3	Flask Server - Clasificarea videoclip-urilor . . . . .	39
3.4	Baze de date . . . . .	39

3.4.1	MongoDB . . . . .	40
3.4.2	Cron Jobs . . . . .	41
3.4.3	Elasticsearch . . . . .	43
3.5	Deployment . . . . .	44
3.5.1	Docker . . . . .	44
3.5.2	Dockerizarea aplicației . . . . .	46
<b>4</b>	<b>Concluzie</b>	<b>47</b>
4.1	Concluzie . . . . .	47
4.2	Perspective . . . . .	48
	<b>Bibliografie</b>	<b>50</b>

# Capitolul 1

## Introducere

### 1.1 Motivație

Motivul pentru care am ales această temă îl reprezintă nevoia de subtitrări pentru videoclipuri, în special pentru filme, dar și pentru tutoriale sau alte tipuri de conținut video, a căror înțelegere este îngreunată de calitatea slabă a sunetului sau de faptul că sunt într-o limbă străină. Astfel, în această lucrare, am ales să abordez problema subtitrărilor prin intermediul unui model de recunoaștere a vorbirii, care extrage audio dintr-un videoclip și generează textul corespunzător.

De asemenea, am ales să abordez și problema căutării videoclipurilor, care constă în căutarea cuvintelor cheie în metadate precum titlu, descriere, topicuri sau chiar în conținutul videoclipului, folosind o bază de date special concepută pentru acest scop, Elasticsearch. [5]

### 1.2 Domenii abordate

Această lucrare abordează 5 domenii principale:

- **Procesarea semnalelor audio** - pentru extragerea audio din videoclipuri și recunoașterea vorbirii
- **Procesarea limbajului natural** - pentru clasificarea videoclipurilor în funcție de conținutul lor
- **Frontend** (*React.js*) pentru interfața cu utilizatorul și pentru a oferi acces la funcționalitățile sistemului
- **Backend** - pentru gestionarea cererilor prin intermediul unui API, cu ajutorul a 3 servicii principale:
  - **Server** (*Node.js*, *Express.js*) pentru gestionarea cererilor și a răspunsurilor

- **Recunoașterea vorbirii** (*Flask*) pentru gestionarea cererilor de recunoaștere a vorbirii
- **Clasificarea videoclipurilor** (*Flask*) pentru clasificarea videoclipurilor în funcție de conținutul lor
- **Baze de date**
  - **MongoDB** pentru stocarea metadatelor videoclipurilor
  - **Elasticsearch** pentru căutarea videoclipurilor în funcție de cuvintele cheie

## 1.3 Structura lucrării

Voi împărți această lucrare în 2 capitole principale, fiecare cu subcapitolele sale.

- **Inteligență artificială** - în care voi aborda *recunoașterea vorbirii* și *clasificarea videoclipurilor* și voi prezenta setul de date folosit, arhitectura modelului, antrenarea și evaluarea acestuia, precum și procesările ulterioare
- **Inginerie software** - în care voi aborda partea de frontend, backend și baze de date, precum și detaliile tehnice ale implementării

# Capitolul 2

## Inteligență artificială

### 2.1 Recunoașterea vorbirii

Pentru a putea recunoaște vorbirea dintr-un videoclip, am ales să folosesc arhitectura *wav2vec 2.0* [2] dezvoltată de Facebook AI Research. Am folosit atât modelul *facebook/wav2vec2-base-960h* antrenat pe setul de date *LibriSpeech* [10], cât și modelul preantrenat *facebook/wav2vec2-base* pe care am continuat să-l antrenez pe seturile de date *Mini LibriSpeech* (subset din LibriSpeech) și *Common Voice Delta Segment 16.1* (subset din Common Voice) [1].

#### 2.1.1 Arhitectura modelului

Modelul *wav2vec 2.0* este un model de învățare profundă alcătuit din 4 componente principale: Latent Feature Encoder (Convolutional Network), Context Network (Transformer Encoder), Quantization Module (Gumbel Softmax) și Contrastive Loss. 2.1

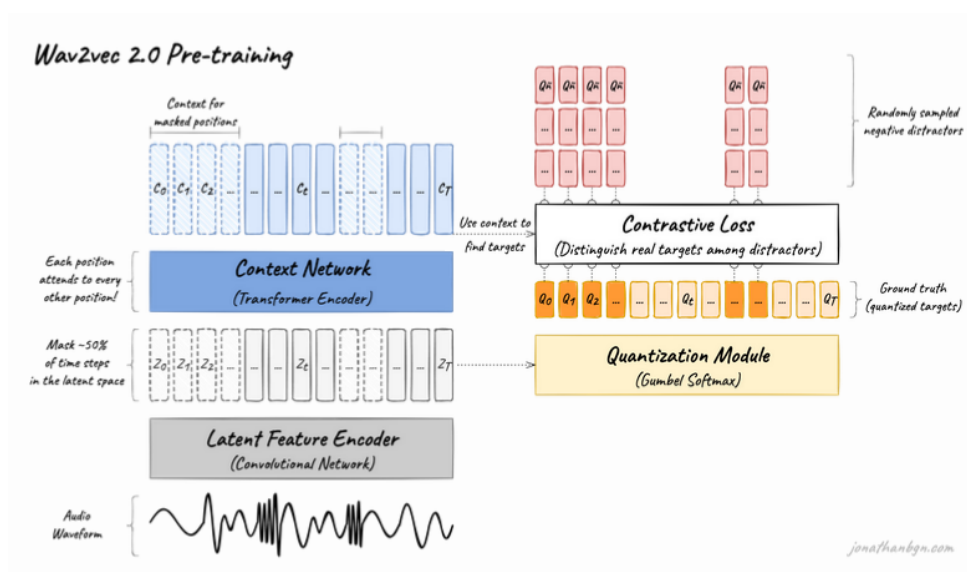


Figura 2.1: Arhitectura modelului *wav2vec 2.0* <sup>1</sup>

## Latent Feature Encoder

Componenta Latent Feature Encoder este o rețea convoluțională care primește ca intrare un semnal audio și aplică o serie de operații de convoluție, normalizare și activări GELU pentru a extrage caracteristici latente din semnalul audio. 2.2

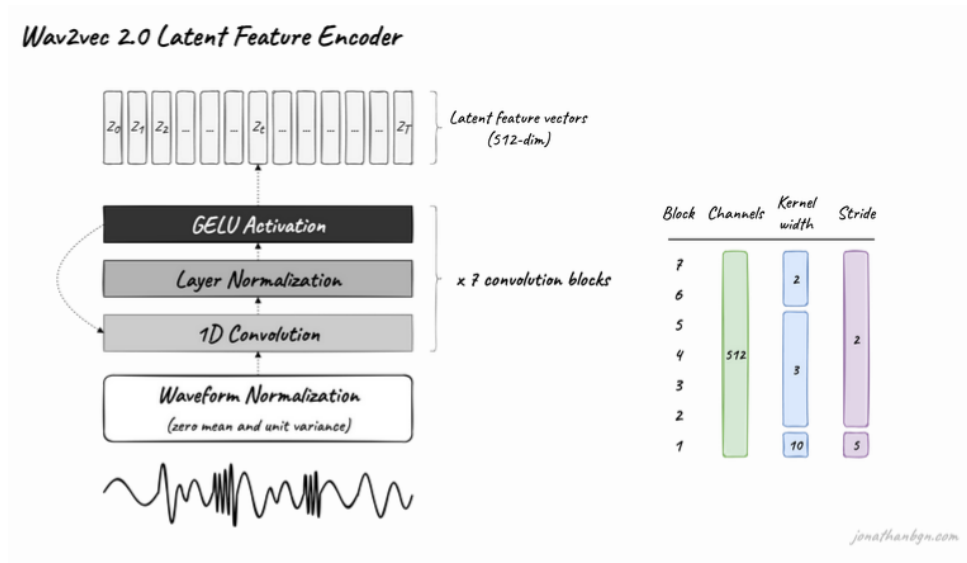


Figura 2.2: Arhitectura componentei Latent Feature Encoder <sup>1</sup>

## Context Network

Componenta Context Network este un encoder de tip Transformer care primește ca intrare caracteristicile latente extrase de componenta Latent Feature Encoder și le procesează pentru a obține o reprezentare contextuală a semnalului audio. Aducând aminte de arhitectura modelului anterior *wav2vec* [12], care folosea tot o rețea convoluțională la acest pas, ar părea că se aseamănă cu componenta anterioară. Diferența constă în faptul că Latent Feature Encoder urmărește să reducă dimensiunea semnalului audio, în timp ce Context Network urmărește să înțeleagă un context mai larg al semnalului audio. 2.3



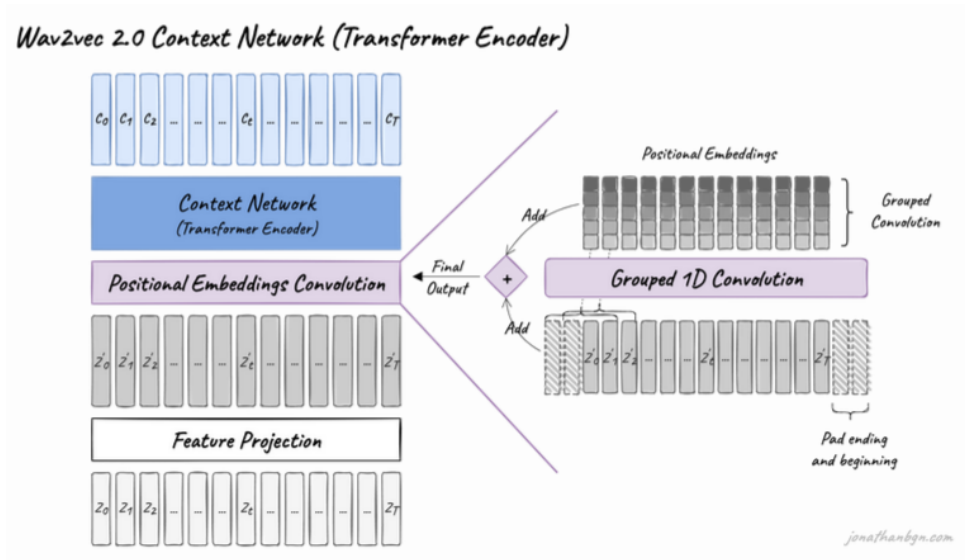


Figura 2.3: Arhitectura componentei Context Network <sup>1</sup>

## Quantization Module

Deoarece modelul *wav2vec 2.0* folosește pentru partea de Context Network un encoder de tip Transformer, ne confruntăm cu problema structurii continue a semnalului audio. Limbajul scris poate fi discretizat într-un set finit de simboluri, în timp ce semnalul audio nu permite în mod direct acest lucru. Astfel, modelul *wav2vec 2.0* folosește un modul de cuantizare care învață automat unități de vorbire din semnalul audio. Intuitiv, se încearcă găsirea unor sunete fonetice finite și reprezentative pentru ieșirile din Latent Feature Encoder. De asemenea, se aplică funcția Gumbel Softmax [9], funcție diferențiabilă care permite antrenarea modelului prin backpropagation. 2.4

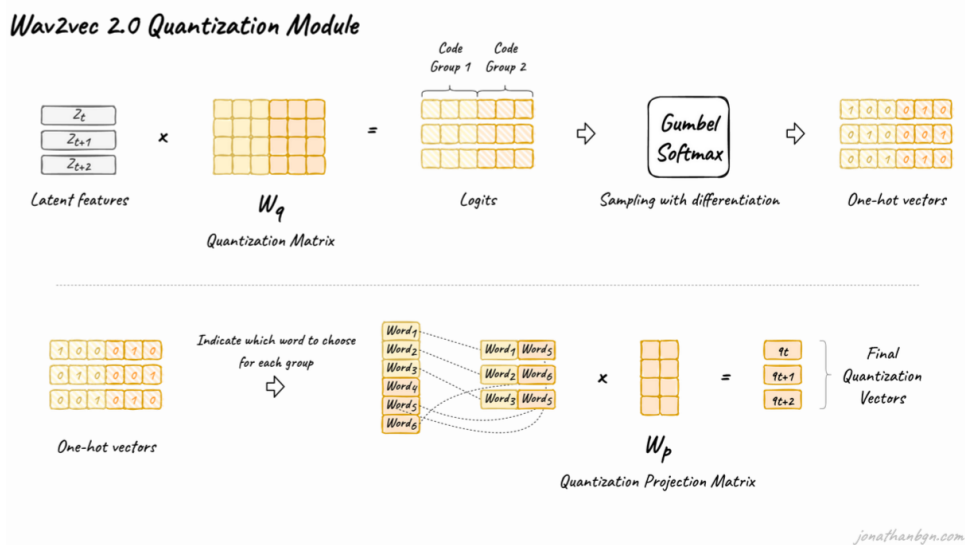


Figura 2.4: Arhitectura componentei Quantization Module <sup>1</sup>

## Contrastive Loss

Pentru antrenarea modelului folosește o mască care ascunde 50% din vectorii proiectați din spațiul latent înainte să fie trecuți prin Context Network. Acest lucru forțează modelul să învețe reprezentări între vectorii proiectați și vectorii ascunși. Pentru fiecare poziție mascată, se alege uniform aleator 100 de exemple negative de la alte poziții și se compară similaritatea cosinus între vectorul proiectat și vectorii aleși. Astfel, funcția de pierdere contrastivă încurajează similaritatea cu exemplele true positive și penalizează similaritatea cu exemplele false positive.

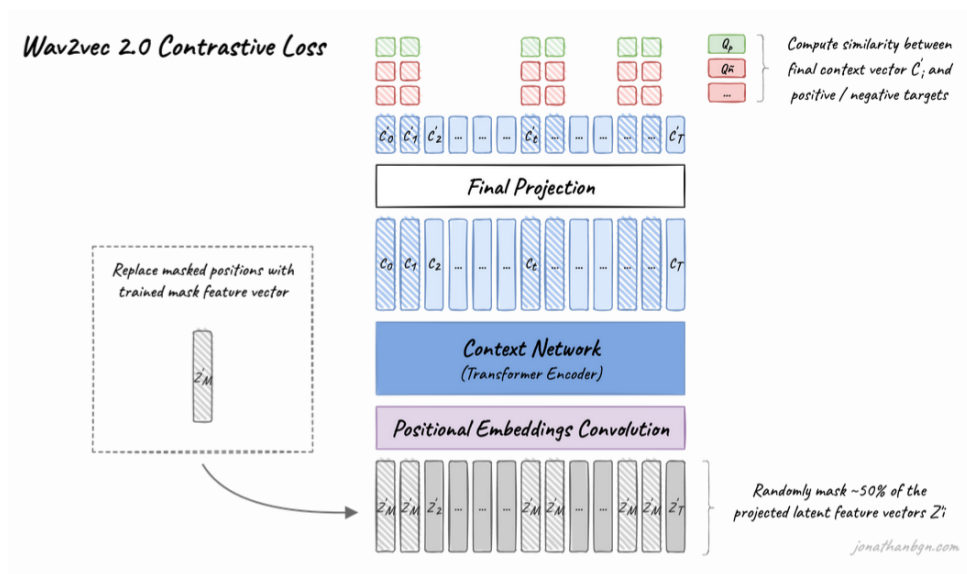


Figura 2.5: Arhitectura componentei Contrastive Loss <sup>1</sup>

### 2.1.2 Setul de date

Modelul oficial a fost preantrenat pe setul de date *LibriSpeech*, iar eu am continuat antrenarea pe seturile de date *Mini LibriSpeech* și *Common Voice Delta Segment 16.1*.

### Mini-LibriSpeech

*Mini LibriSpeech* este un subset al setului de date *LibriSpeech* care conține aproximativ 2 ore de înregistrări audio la o frecvență de eșantionare de 16 kHz. În medie, fiecare înregistrare are o durată de 6.72 secunde, cel mai lung audio având o durată de 31.5 secunde.

<sup>1</sup>Imaginile au fost preluate de pe site-ul lui Jonathan Bgn, "Illustrated Wav2Vec 2.0", disponibil la: <https://jonathanbgn.com/2021/09/30/illustrated-wav2vec-2.html>.

## Common Voice Delta Segment 16.1

*Common Voice Delta Segment 16.1* este un subset al setului de date *Common Voice* care conține aproximativ 2 ore de înregistrări audio la o frecvență de eșantionare de 48 kHz. A fost nevoie să reduc frecvența de eșantionare la 16 kHz pentru a putea folosi aceste date la antrenarea modelului. În medie, fiecare înregistrare are o durată de 5.63 secunde, cel mai lung audio având o durată de 10.47 secunde.

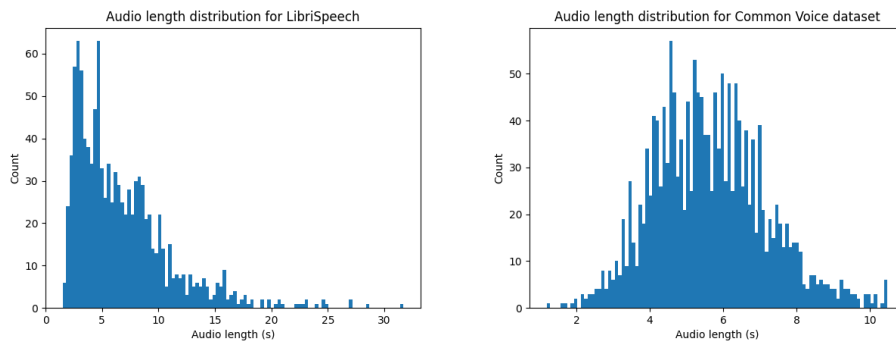


Figura 2.6: Distribuția duratelor secvențelor audio din seturile de date *Mini LibriSpeech* și *Common Voice Delta Segment 16.1*

## Concluzie

Menționez aceste detalii deoarece pentru generarea subtitrărilor vom avea nevoie de audio-uri mult mai lungi decât cele folosite pentru antrenare care nu ar încăpea în memorie. Astfel, va trebui să folosim o tehnică de segmentare a secvențelor audio în bucăți mai mici pentru a putea fi procesate. Mai multe detalii despre această tehnică vor fi prezentate în secțiunea *Subtitrări*. 3.3.2

### 2.1.3 Antrenarea modelului

Pentru antrenarea modelului am folosit limbajul de programare Python și biblioteca Hugging Face care pune la dispoziție o serie de pachete precum *transformers* și *datasets*. Hiperparametrii folosiți pentru fine-tuning-ul modelului sunt:

- **Learning rate** - pentru a controla cât de mult se modifică gradientii în timpul antrenării
- **Weight decay** - pentru a controla cât de mult se penalizează valorile mari ale parametrilor
- **Warmup steps** - pentru a controla cât de mult se modifică learning rate-ul în primele pași ai antrenării

- **Batch size** - câte exemple se procesează în același timp

Hiperparametru	Valoare
Rata de învățare ( <i>learning rate</i> )	$1 \times 10^{-4}$
Descrescerea greutatei ( <i>weight decay</i> )	0.005
Pași de încălzire ( <i>warmup steps</i> )	1000
Dimensiunea lotului ( <i>batch size</i> )	4

Tabela 2.1: Hiperparametrii folosiți pentru fine-tuning-ul modelului *wav2vec2*

Am antrenat modelul pe seturile de date *Mini LibriSpeech* și *Common Voice Delta Segment 16.1* pe o placă grafică NVIDIA Tesla V100 pusă la dispoziție de Google Colab. Am făcut un checkpoint la fiecare 500 de pași pentru a putea monitoriza evoluția modelului. Rezultatele checkpoint-urilor pentru modelul *facebook/wav2vec2-base* sunt prezentate în cele două tabele de mai jos.

Tabela 2.2: Mini LibriSpeech

Step	Train loss	Val loss
500	3.840	3.099
1000	1.202	0.586
1500	0.360	0.352
2000	0.231	0.333
2500	0.163	0.357
3000	0.136	0.331
3500	0.114	0.369
4000	0.104	0.348
4500	0.094	0.335
5000	0.083	0.284
5500	0.078	0.332
6000	0.072	0.356
6500	0.069	0.393
7000	0.066	0.380

Tabela 2.3: Common Voice Delta 16.1

Step	Train loss	Val loss
500	3.919	3.236
1000	1.287	0.570
1500	0.368	0.400
2000	0.226	0.371
2500	0.166	0.402
3000	0.136	0.475
3500	0.116	0.445
4000	0.097	0.448
4500	0.096	0.404
5000	0.079	0.459
5500	0.080	0.400
6000	0.069	0.445
6500	0.073	0.421
7000	0.064	0.440

## Note

Se observă că modelul a început să învețe destul de repede, scăzând pierderea de antrenare de la 3.840 la 0.066, respectiv de la 3.919 la 0.064 în doar 7000 de pași. Urmărind graficul, se observă fenomenul de *overfitting* care apare în jurul pașilor 5000-6000, motiv pentru care am ales să opresc antrenarea la 7000 de pași și să folosesc checkpoint-ul de la pasul 5000, respectiv 5500.

### 2.1.4 Îmbunătățire cu n-gram language model

Pentru a îmbunătăți recunoașterea vorbirii, am folosit un n-gram language model care mărește performanța modelului de la **wer 4.2%** la **wer 2.9%**, aducând o îmbunătățire de **1.3%**.

#### N-gram Language Model

Un n-gram language model este un model statistic care estimează, în cazul nostru, probabilitatea apariției unui caracter având în vedere cele n-1 caractere anterioare. Modelul se bazează pe ipoteza Markov de ordinul n, conform căreia putem aproxima probabilitatea apariției unui caracter folosind doar ultimele n caractere. Formula de mai jos ilustrează această idee.

$$P(w_1, w_2, \dots, w_n) = \prod_{i=1}^n P(w_i | w_1, w_2, \dots, w_{i-1}) \approx \prod_{i=1}^n P(w_i | w_{i-(n-1)}, \dots, w_{i-1}) \quad (2.1)$$

Am folosit un context de 5 caractere și am antrenat modelul pe setul de date *Helsinki-NLP/europarl* [13] deoarece conține și texte în limba engleză și putem avea certitudinea că textele sunt corecte din punct de vedere gramatical. Cu ajutorul librăriei *KenLM* [8], am antrenat modelul de n-grame și apoi am creat un procesor specific Hugging Face pentru modelele *wav2vec 2.0*.

În mod normal, modelul *wav2vec 2.0* ia argumentul maxim din distribuția de probabilitate pentru a prezice caracterul următor. În schimb, cu ajutorul n-gram language model, aceste probabilități sunt alterate pentru a se apropia de limbajul natural.

#### Îmbunătățire cu Transformer Language Model

Având în vedere performanța arhitecturii Transformer în învățarea secvențelor, o altă abordare ar fi utilizarea unui model de limbaj bazat pe Transformer. Paper-ul original [2] menționează în Appendix-ul C că un model de limbaj bazat pe Transformer este într-adevăr mai bun decât un model de limbaj bazat pe n-grame, dar durata antrenării și a inferenței este mult mai mare.

Tabelul de mai jos a fost extras din paper-ul original și prezintă rezultatele obținute.

2.7

Model	Unlabeled data	LM	dev		test	
			clean	other	clean	other
LARGE - from scratch	-	None	2.8	7.6	3.0	7.7
	-	4-gram	1.8	5.4	2.6	5.8
	-	Transf.	1.7	4.3	2.1	4.6
BASE	LS-960	None	3.2	8.9	3.4	8.5
		4-gram	2.0	5.9	2.6	6.1
		Transf.	1.8	4.7	2.1	4.8
LARGE	LS-960	None	2.6	6.5	2.8	6.3
		4-gram	1.7	4.6	2.3	5.0
		Transf.	1.7	3.9	2.0	4.1
LARGE	LV-60k	None	2.1	4.5	2.2	4.5
		4-gram	1.4	3.5	2.0	3.6
		Transf.	1.6	3.0	1.8	3.3

Figura 2.7: Rezultatele obținute cu un model simplu, un model de limbaj bazat pe n-grame și un model de limbaj bazat pe Transformer

## 2.2 Clasificarea videoclipurilor

Videoclip-urile încărcate pe platform vin însoțite de metadate precum: titlu, descriere și, folosind modelul prezentat anterior, subtitrări. Pentru o experiență mai personalizată, am ales să clasific videoclipurile în funcție de subiectul abordat în topicuri precum: politics, sport, entertainment tehnologie și afaceri. Am folosit un modelul de clasificare *BERT*, Bidirectional Encoder Representations from *Transformers*, dezvoltat de *Google* [4] pe care l-am antrenat pe setul de date *BBC News* [7].

### 2.2.1 Arhitectura modelului

Modelul *BERT* este un model de învățare profundă care are la bază partea de encoder a unui *Transformer* [14], scopul lui fiind de a înțelege contextul cuvintelor într-o propoziție. Modelul *BERT* vine în două variante: *BERT-base* și *BERT-large*, cele două diferă prin numărul de blocuri de encoder (12 și 24), numărul de neuroni din stratul fully-connected (768 și 1024) și numărul de attention heads (12 și 16), dar conceptual sunt la fel.

Spre deosebire de arhitectura standard a unui *Transformer*, *BERT* adaugă un token special la începutul fiecărei propoziții, *[CLS]*, folosit pentru clasificare. Intuitiv, acest token va reține informații despre întreaga propoziție și va putea fi folosit pentru rețeaua de clasificare care va determina topicul propoziției. 2.8

Ieșirile modelului *BERT* au dimensiunea 768, iar pentru partea de clasificare sunt trecute prin un strat fully-connected cu 5 neuroni, câte unul pentru fiecare clasă. Cu ajutorul funcției *softmax*, se obține o distribuție de probabilitate peste cele 5 clase, iar clasa cu cea mai mare probabilitate este cea aleasă.

<sup>2</sup>Imaginea a fost preluată de pe site-ul lui Jay Alammar, “The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning)”, disponibil la: <https://jalamar.github.io/illustrated-bert/>.

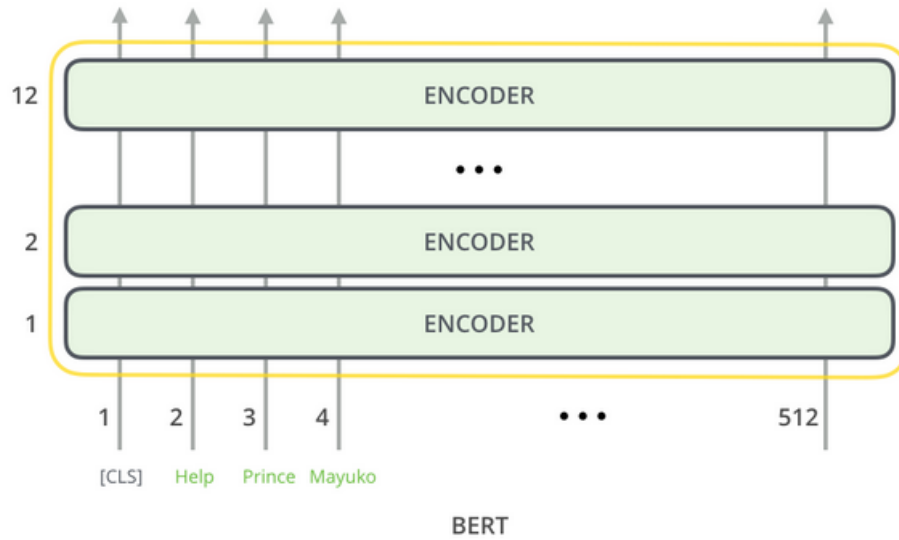


Figura 2.8: Arhitectura modelului *BERT* <sup>2</sup>

### 2.2.2 Setul de date

Pentru antrenarea modelului am folosit setul de date *BBC News* care conține 2225 de articole între anii 2004-2005. Setul de date este împărțit în 5 clase: politics, sport, entertainment, technology și business. Mai jos am prezentat distribuția datelor din setul de date din punct de vedere al exemplelor pe clasă și al lungimii medii a articolelor. 2.9

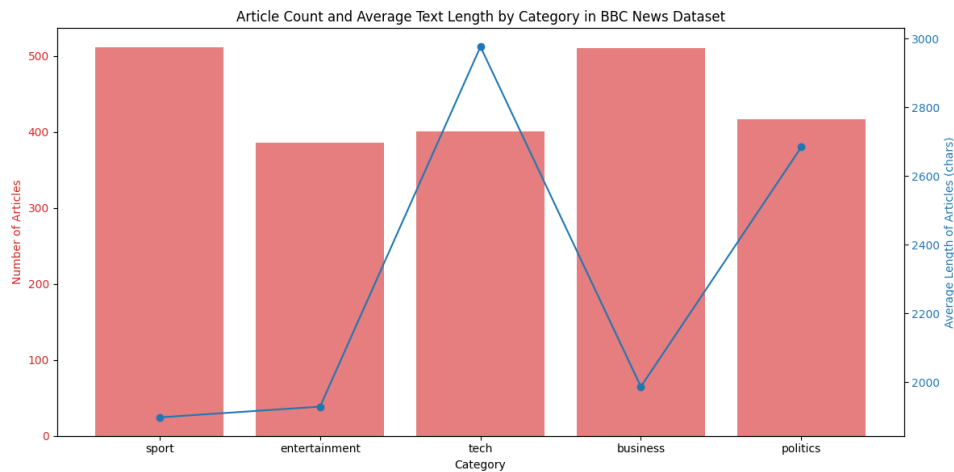


Figura 2.9: Distribuția claselor din setul de date *BBC News*

## 2.2.3 Antrenarea modelului

Antrenarea modelului a fost făcută folosind limbajul de programare Python, biblioteca PyTorch. [11] și plecând de la modelul preantrenat *bert-base-uncased* pus la dispoziție de Hugging Face. Codul a fost structurat în:

- **dataset.py** - creează un Dataset specific PyTorch pentru setul de date *BBC News*, implementând metodele `__len__` și `__getitem__`; metoda `__getitem__` aplică tokenizer-ul pe text cu lungimea maximă de 512 token-uri și returnează un tuplu de `input_ids`, `attention_mask` și `label`
- **model.py** - definește arhitectura modelului *BERT* și a rețelei de clasificare: inițializează modelul preantrenat *bert-base-uncased*, extrage token-ul special *[CLS]* și adaugă un strat liniar cu 5 neuroni pentru clasificare
- **trainer.py** - antrenează modelul pe setul de date *BBC News* folosind hiperparametrii din primii ca parametru în linia de comandă și salvează modelul de fiecare dată când obține o acuratețe mai bună pe setul de validare

În figurile de mai jos 2.10 sunt ilustrate pierderile de antrenare și de evaluare pentru modelul *BERT* folosind ca rată de învățare  $1 \times 10^{-4}$  și  $1 \times 10^{-5}$ . Se observă fenomenul de *overfitting* pentru rata de învățare  $1 \times 10^{-4}$ , motiv pentru care am ales să experimentez cu valori mai mici.

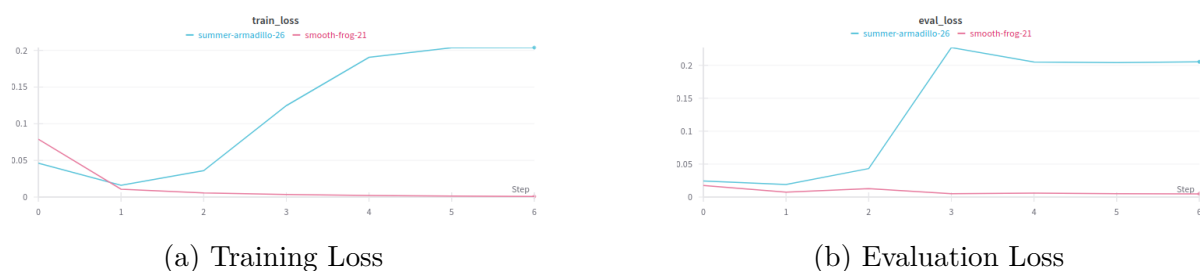


Figura 2.10: Training and Evaluation Loss Graphs

Am încercat mai multe experimente plecând de la aceleași hiperparametri, cu rata de învățare  $1 \times 10^{-5}$ , pentru a vedea cum se comportă modelul la diferite inițializări ale ponderilor. În figura 2.11 sunt ilustrate valorile obținute pentru acuratețea modelului pe setul de validare.

### Hiperparametrii

Pentru modelul final pe care l-am folosit pentru clasificarea videoclipurilor am ales următorii hiperparametrii:



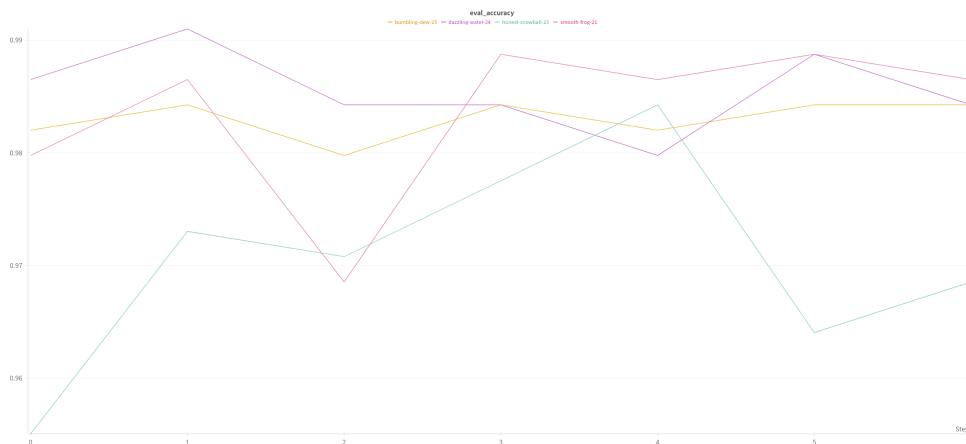


Figura 2.11: Evaluation Accuracy Graph

Hiperparametru	Valoare
Rata de învățare ( <i>learning rate</i> )	0.00001
Dimensiunea lotului ( <i>batch size</i> )	8
Optimizator ( <i>optimizer</i> )	Adam
Funcția de pierdere ( <i>loss function</i> )	CrossEntropyLoss

Tabela 2.4: Hiperparametrii modelului *BERT*

## 2.3 Concluzie

Cele două modele prezentate anterior, *wav2vec 2.0* și *BERT*, au fost încărcate pe platforma Huggingface și sunt disponibile în pachetul *transformers*.

### wav2vec 2.0

Utilizarea modelului de recunoaștere a vorbirii necesită folosirea modulului *Wav2Vec2ForCTC* și a modulului *Wav2Vec2Processor* sau *Wav2Vec2ProcessorWithLM* pentru a folosi varianta îmbunătățită cu n-grame. Codul de mai jos ilustrează cum se pot folosi modelele prezentate anterior.

```

1  from transformers import Wav2Vec2ForCTC, Wav2Vec2Processor,
   Wav2Vec2ProcessorWithLM
2
3  asr_repos = {
4      "minilibrispeech": "3funnn/wav2vec2-base-minilibrispeech",
5      "common-voice": "3funnn/wav2vec2-base-common-voice",
6  }
7  lm_boosted_repos = {
8      "minilibrispeech": "3funnn/wav2vec2-base-minilibrispeech-lm",
9      "common-voice": "3funnn/wav2vec2-base-common-voice-lm",
10 }
11

```

```

12 repo_name = "minilibrispeech"
13 boost_lm = True
14 model = Wav2Vec2ForCTC.from_pretrained(asr_repos[repo_name])
15
16 if boost_lm:
17     processor = Wav2Vec2ProcessorWithLM.from_pretrained(
18 lm_boosted_repos[repo_name])
19 else:
20     processor = Wav2Vec2Processor.from_pretrained(asr_repos[repo_name])

```

## BERT

Modelul de clasificare a videoclipurilor este disponibil în pachetul *transformers* sub numele *3funnn/bert-topic-classification*. Modelul primește ca intrare un text care poate fi o concatenare a titlului și descrierii videoclipului și returnează o distribuție de probabilitate pentru fiecare din cele 5 clase. Huggingface pune la dispoziție funcția *pipeline* care permite folosirea modelului într-un mod simplu. Codul de mai jos ilustrează acest lucru.

```

1 from transformers import pipeline
2
3 classifier = pipeline("text-classification", model="3funnn/bert-topic-
4 classification")
5
6 text = "This is a video about the latest technology in AI."
7 result = classifier(text)
8 print(result)
# Output: {'label': 'tech', 'score': 0.897}

```

# Capitolul 3

## Inginerie software

### 3.1 Arhitectura aplicației

Am ales să folosesc arhitectura MERN (MongoDB, Express.js, React, Node.js) la care am mai adăugat două servere de Flask pentru recunoașterea vorbirii și clasificarea videoclip-urilor, o bază de date Elasticsearch pentru căutare și Kibana pentru vizualizarea datelor din Elasticsearch.

În figura de mai jos 3.1 este prezentată arhitectura aplicației.

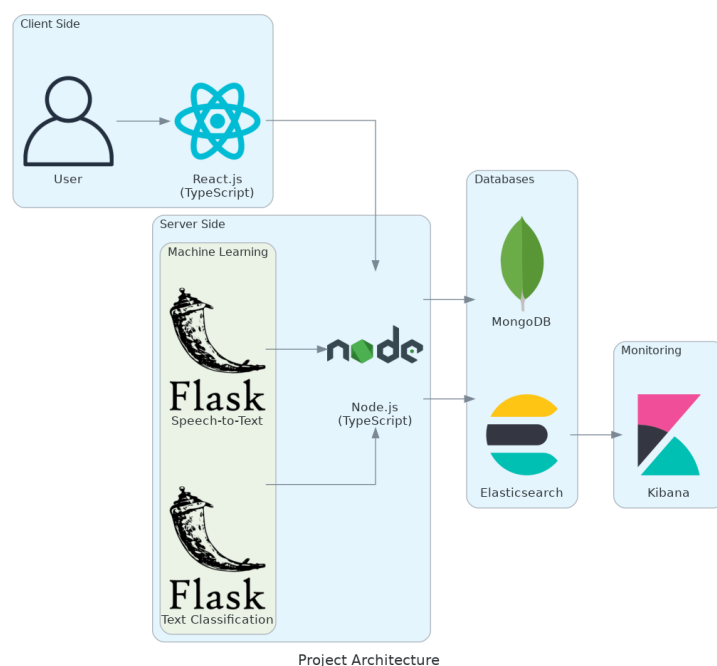


Figura 3.1: Arhitectura aplicației

## 3.2 Frontend

Am folosit React și limbajul de programare TypeScript pentru interfața grafică a aplicației web.

React este o bibliotecă JavaScript care simplifică procesul de dezvoltare a aplicațiilor web prin caracteristici specifice precum reutilizarea componentelor și actualizarea independentă a elementelor interfeței grafice. TypeScript este un limbaj de programare creat de Microsoft care extinde limbajului JavaScript cu tipuri statice, aspect util care îmbunătățește calitatea codului.

### 3.2.1 React

Librăria React impune o serie de concepte și bune practici care abstractizează și modularizează codul aplicației web. Dintre acestea, cele mai importante sunt:

- **Components:** Ideea de bază a bibliotecii React o reprezintă componentele, elemente reutilizabile care încapsulează și izolează logica codului.
- **Props:** Proprietățile sunt parametri transmiși de la componenta părinte la componenta copil, fiind folosite pentru customizarea comportamentului componentelor.
- **State:** Starea reprezintă datele interne ale componentelor, fiind folosite pentru actualizarea interfeței grafice din JSX.
- **Hooks:** Hooks sunt funcții speciale care permit interacțiunea cu starea și ciclul de viață al componentelor, fiind succesorul claselor din versiunile mai vechi ale bibliotecii.
- **Contexts:** Contextele sunt o modalitate de a controla transmiterea datelor între componente fără a folosi proprietăți, intuitiv, creând un fel de scope pentru componentele din interiorul contextului.
- **Models:** Deoarece folosim TypeScript, avem nevoie de modele pentru a defini tipurile de date folosite în aplicație.
- **Views/Pages:** Pot fi considerate componente principale, fiind rutele aplicației web.
- **Services:** Serviciile sunt funcții asincrone care primesc date din frontend, creează cereri HTTP către server (folosind Axios sau Fetch) și returnează răspunsurile primite de la server.
- **Utils:** Funcții utilitare care nu sunt legate de o componentă anume.

### 3.2.2 Autentificare

Autentificarea utilizatorilor nu folosește niciun serviciu extern, ci se bazează pe un sistem propriu de înregistrare și logare. Fiecărei cereri de la client i se atașează un token JWT (JSON Web Token) generat de server la logare/înregistrare, fiind folosit pentru a verifica identitatea utilizatorului.

#### JWT

JSON Web Token (JWT) encodează în structura sa 3 componente: header, payload și signature. Header-ul stochează tipul de token și algoritmul de criptare, payload-ul conține informații despre utilizator precum numele și data emiterii, iar signature-ul este un hash al primelor două componente la care se mai adaugă un secret.

#### Înregistrare și logare

Atât înregistrarea 3.2 cât și logarea 3.3 sunt realizate prin intermediul unor formulare care cer informații precum numele, prenumele, adresa de email, un nume de utilizator, o parolă și poză de profil. Pentru aceste formulare au fost folosite validatoare care verifică dacă datele introduse sunt corecte.

Odată ce datele sunt introduse, sunt validate încă o dată la nivel de server pentru a evita eventualele atacuri și, dacă sunt corecte, se generează un token JWT care este trimis înapoi la client. Acest token este salvat în memoria locală a browser-ului, dar și în contextul utilizatorului pentru a fi folosit în cererile ulterioare către server.

Pentru deconectare, token-ul este șters din memoria locală și contextul utilizatorului.

#### Profil


Pagina de profil prezintă informații despre utilizator precum numele, prenumele, numele de utilizator, adresa de email și poză de profil și îi permite acestuia să le actualizeze cu ajutorul unui formular.

### 3.2.3 Video

Platforma permite utilizatorilor să încarce videoclip-uri, să le vizualizeze și să interacționeze cu acestea prin intermediul unor funcții precum aprecieri și comentarii.

#### Încărcare videoclip

Încărcarea videoclip-urilor se face print-un buton care redirecționează utilizatorul către pagina de încărcare. Aici, utilizatorul poate selecta un fișier video și adăuga un titlu și



## Register

First Name \*

Robert

Last Name \*

Trifan

Email Address \*

robert.trifan@gmail.com

Username \*


robert.trifan

Password \*

\*\*\*\*\*

SIGN UP

[Already have an account? Sign in](#)



## Login

Username \*

robert.trifan

Password \*


\*\*\*\*\*

LOG IN

[Don't have an account? Sign up](#)

Figura 3.3: Pagina de logare

Figura 3.2: Pagina de înregistrare



Username

robert.trifan

Email

robert.trifan@gmail.com


First Name

Robert

Last Name

Trifan

EDIT PROFILE



Username

robert.trifan

Email

robert.trifan@gmail.com

First Name

Robert-Gabriel

Last Name

Trifan

SAVE CHANGES

CANCEL

Figura 3.4: Pagina de profil

Figura 3.5: Editare profil

o descriere. După ce videoclip-ul este încărcat, acesta este trimis către server pentru a fi prelucrat (stocarea în baza de date, extragerea subtitrărilor, clasificarea videoclip-ului).

## Vizualizare

Videoclip-urile încărcate sunt afișate pe pagina principală a aplicației web sub forma unui grid, fiecare videoclip având un thumbnail, un titlu și numele autorului. Atunci când este accesat un videoclip, utilizatorul este redirecționat către o pagină dedicată unde poate vizualiza videoclip-ul și interacționa cu acesta prin intermediul apreriilor și comentariilor.

### Upload Video

Title \*

Description \*

Video File  
 +

UPLOAD

Figura 3.6: Pagina de încărcare a videoclip-urilor

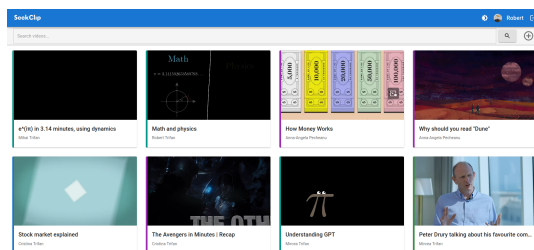


Figura 3.7: Grid de videoclip-uri

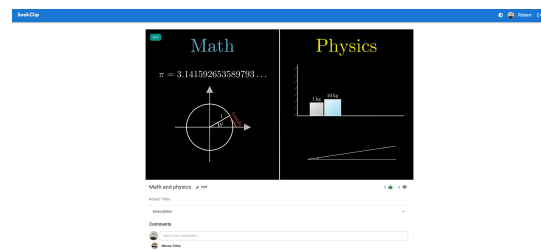


Figura 3.8: Vizualizare videoclip

## Editare

Autorii videoclip-urilor încărcate pot accesa pagina de editare a videoclip-ului prin butonul de editare de pe pagina de vizualizare. Fiind o rută protejată, utilizatorul trebuie să fie autentificat și să fie autorul videoclip-ului pentru a putea schimba metadatele videoclip-ului.

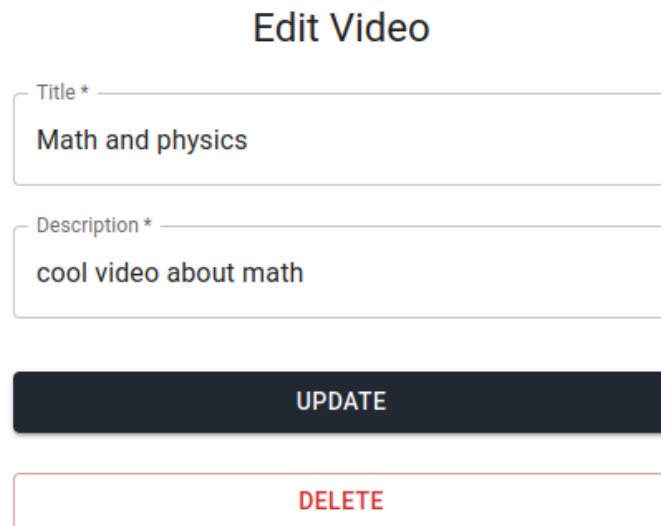
## Aprecieri & Comentarii

Pagina de vizualizare a videoclip-ului permite utilizatorului să dea like, dislike videoclip-ului sau să își retragă aprecierea și să adauge comentarii.

Fiecare videoclip reține o listă cu id-urile utilizatorilor care au dat like sau dislike și o listă cu comentariile adăugate. De asemenea, utilizatorii pot răspunde la comentarii, creând astfel o ierarhie de comentarii și pot edita sau șterge comentariile proprii.

Dacă un comentariu are răspunsuri, atunci textul devine [deleted] fiind un *soft-delete*. Altfel, dacă un comentariu nu are răspunsuri, atunci acesta este șters complet din baza de date fiind considerat un *hard-delete*. Comentariile sunt sortate după data adăugării, cele mai recente fiind afișate mai sus.

Trebuie să menționez un detaliu important legat de ierarhia comentariilor. Ar fi

The image shows a web form titled "Edit Video". It contains two text input fields. The first field is labeled "Title \*" and contains the text "Math and physics". The second field is labeled "Description \*" and contains the text "cool video about math". Below these fields are two buttons: a dark grey button labeled "UPDATE" and a light red button labeled "DELETE".

**Edit Video**

Title \*  
Math and physics

Description \*  
cool video about math

UPDATE

DELETE

Figura 3.9: Editare videoclip

foarte costisitor, din punct de vedere al performanței, ca de fiecare dată când se șterge un comentariu să se reconstruiască arborele de comentarii. De aceea, am creat un serviciu care rulează o dată pe zi, construiește ierarhia comentariilor și șterge acele drumuri în arbore care nu mai sunt folosite. Mai multe detalii despre acest serviciu se găsesc în secțiunea 3.4.2.

### 3.2.4 Căutare

#### Bara de căutare

Pe pagina principală a aplicației web se află o bară de căutare care permite utilizatorului introducerea unor cuvinte cheie specifice videoclip-urilor pe care le caută. 3.11.

Când utilizatorul introduce textul în bara de căutare, aplicația web trimite o cerere către server care caută videoclip-urile într-o bază de date Elasticsearch. Rezultatele sunt afișate pe aceeași pagină, sub forma unui grid asemănător cu cel de pe pagina principală.

### 3.2.5 Temă

Întreaga aplicație web poate fi personalizată prin intermediul a două teme: light și dark. Utilizatorul poate schimba tema din navbar cu ajutorul unui buton care schimbă valoarea unei variabile din contextul temei și propagă această schimbare în întreaga aplicație web.



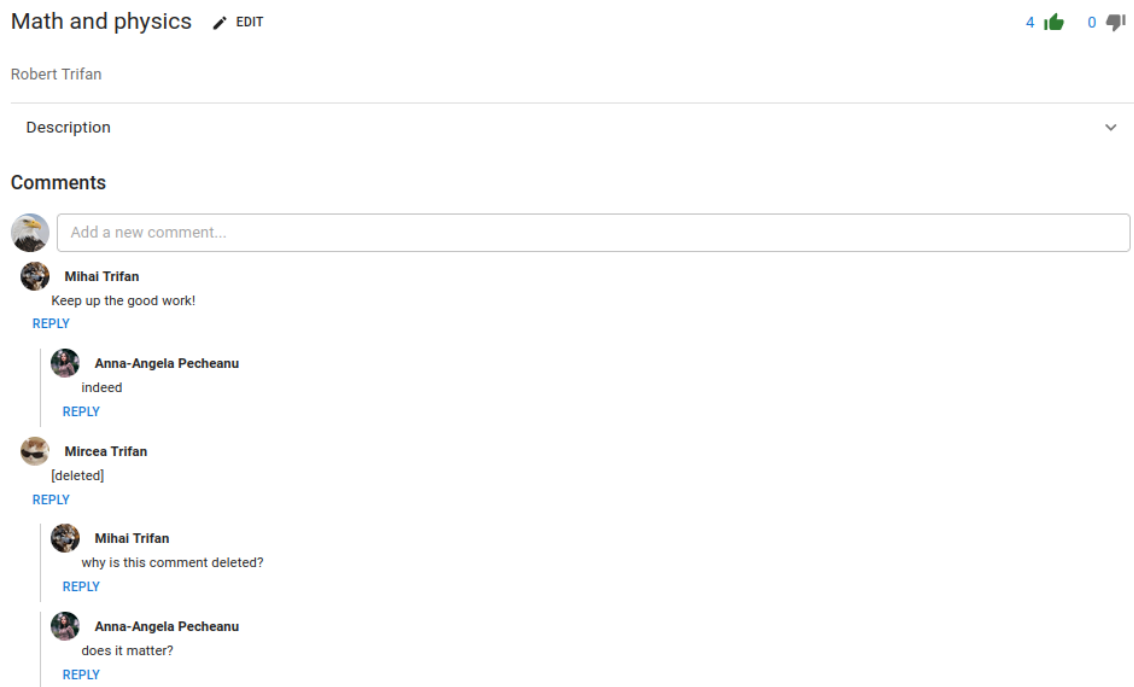


Figura 3.10: Interacțiunea cu videoclip-ul

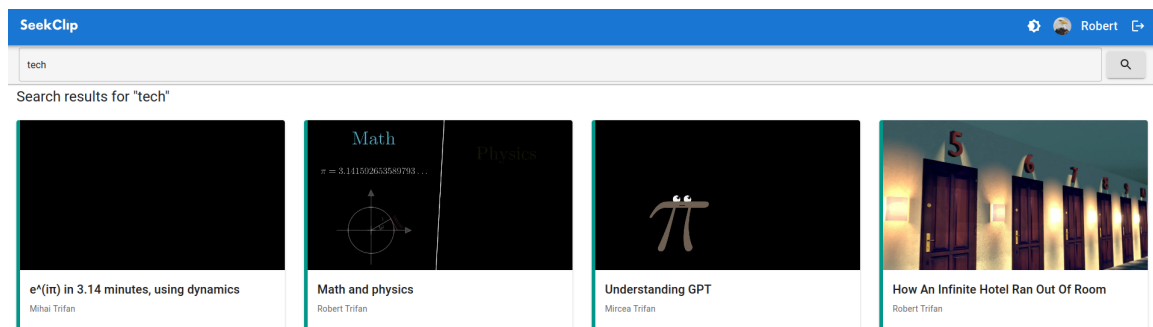


Figura 3.11: Bară de căutare

### 3.2.6 Rute protejate

Pentru a interzice accesul la anumite pagini din motive de securitate, am folosit un sistem de rute protejate:

- **User Protected:** se asigură că sunt accesate doar de utilizatori autentificați, în caz contrar fiind redirecționați către pagina de logare; de exemplu, pagina de profil, încărcarea videoclip-urilor, postarea de comentarii/aprecieri.
- **Video Owner Protected:** se asigură că sunt accesate doar de autorii videoclip-urilor, și implicit de utilizatorii autentificați, în caz contrar fiind redirecționați către pagina principală; de exemplu, editarea videoclip-urilor.

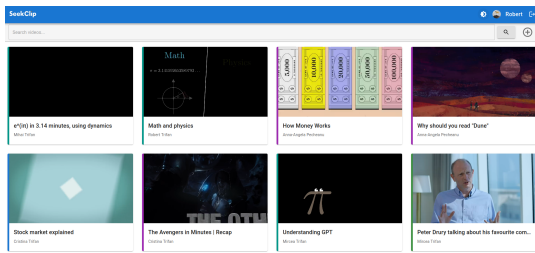


Figura 3.12: Tema light

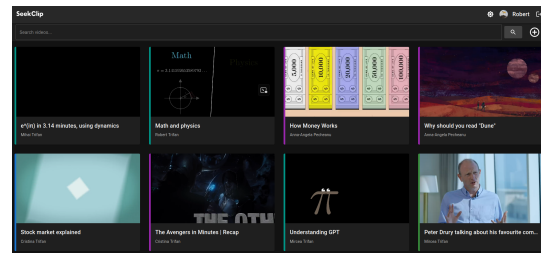


Figura 3.13: Tema dark

## 3.3 Backend

### 3.3.1 Express Server - Procesarea cererilor

Am creat un server Node.js folosind Express.js în limbajul de programare TypeScript care gestionează cererile primite de la client și le rutează către serviciile corespunzătoare.

Astfel, codul este structurat:

- **Routes:** definesc rutele serverului și asociază cererile primite cu funcțiile corespunzătoare din controlere.
- **Controllers:** conțin funcțiile care procesează cererile primite de la client și apelează serviciile necesare.
- **Middlewares:** funcții care se execută între primirea cererilor și procesarea acestora, adăugând/verificând informații suplimentare din cererea primită.
- **Models:** definește structura entităților folosite în baza de date MongoDB și a interfețelor pentru a asigura tipurile de date.
- **Utils:** funcții utilitare care facilitează procesarea cererilor.
- **Config:** inițializează comunicarea cu alte servicii precum baza de date Elasticsearch.

## REST API

Comunicarea între client și server se face prin intermediul unui REST API. Prin definiție, REST (Representational State Transfer) impune existența a 3 componente: client, server și resurse.

Clientul îl reprezintă aplicația web care face cereri către server, serverul este aplicația care primește gestionează cererile și returnează răspunsuri, iar resursele sunt datele stocate în baza de date pe care serverul le manipulează.

Conceptul de REST API are la bază 6 principii:

- **Separarea client-server:** clientul și serverul sunt entități separate și izolate, serverul nu poate face cereri către client, doar clientul poate face cereri către server; în acest fel, cele două entități pot evolua independent.

- **Uniform Interface:** având în vedere diversitatea limbajelor de programare folosite atât pentru client cât și pentru server, este necesară o interfață comună, independentă de limbaj care să faciliteze comunicarea între cele două entități; astfel REST API este construit peste protocolul HTTP și folosește metodele standardizate GET, POST, PUT, DELETE.
- **Stateless:** fiecare cerere primită de la client conține toate informațiile necesare pentru a fi procesată cu succes, serverul nefiind nevoit să păstreze informații despre cererile anterioare.
- **Layered System:** cererea primită de la client poate trece prin mai multe straturi intermediare folosite pentru a îmbunătăți performanța, securitatea sau scalabilitatea aplicației.
- **Cacheable:** cererile anterioare pot fi stocate în cache pentru a fi refolosite la cereri ulterioare, reducând astfel timpul de răspuns și traficul de date între client și server.
- **Code on Demand (Optional):** serverul poate trimite cod executabil către client pentru a fi rulat în browser, însă această funcționalitate este opțională

## Codurile de stare HTTP

Pentru a comunica rezultatul unei cereri, există un standard de coduri HTTP împărțit în următoarele 5 categorii:

- **1xx:** informații, cererea a fost primită, se continuă procesarea.
- **2xx:** succes, cererea a fost primită, procesată și răspunsul a fost generat cu succes.
- **3xx:** redirectionare, clientul trebuie să facă o altă cerere pentru a obține răspunsul.
- **4xx:** erori de la client, cererea a fost greșită sau nu poate fi procesată.
- **5xx:** erori de la server, serverul a întâmpinat o problemă în procesarea cererii.

## Routes

Rutele sunt construite recursiv, plecând de la rădăcină și adăugând pentru fiecare API un nou nivel de rute. În cadrul aplicației noastre, am definit următoarele rute:

- **/api/auth:** rutele responsabile de autentificarea utilizatorilor, care mai apoi se specializează în /api/auth/register și /api/auth/login.
- **/api/user:** rutele responsabile de gestionarea informațiilor utilizatorilor, se specializează în /api/user/:username și /api/user/update/:username.

- **/api/video:** rutele responsabile de gestionarea videoclip-urilor, se specializează în operațiile CRUD (Create, Read, Update, Delete) pentru videoclipuri și a interacțiunilor cu acestea.
- **/api/comment:** rutele responsabile de gestionarea comentariilor, aduc funcționalități precum adăugarea, editarea și ștergerea comentariilor.

## Middlewares

Middleware-urile sunt funcții care se execută între primirea cererii și procesarea acesteia, de obicei adăugând sau verificând informații suplimentare din cererea primită. În cadrul aplicației noastre, am folosit două middleware-uri:

- **Auth Middleware:** se adaugă la cererile care necesită ca utilizatorul să fie autentificat, verificând astfel existența token-ului JWT și dacă acesta este valid. Verificarea se face prin decriptarea token-ului și verificarea semnăturii cu ajutorul secretului stocat pe server. Fiecare cerere are în header-ul ei un câmp *Bearer Token* care conține token-ul JWT. Odată decriptat, middleware-ul adaugă în corpul cererii informațiile despre utilizatorul autentificat.
- **Multer Middleware:** se folosește pentru cererile care necesită încărcarea unor fișiere de tip imagine sau videoclip; middleware-ul se asigură că fișierul are formatul corect (de exemplu, imaginea este de tip .jpg/.png/.jpeg și videoclip-ul este de tip .mp4/.mov/.avi). Acesta creează un director în interiorul căruia se salvează fișierul încărcat și adaugă calea către fișier în baza de date. Serverul expune apoi url-ul către id-ul fișierului încărcat pentru a putea fi accesat de client.

## Models

Modelele sunt folosite pentru a defini structura entităților folosite în baza de date MongoDB. Deci, pentru fiecare entitate din baza de date, definim o *Schema* care conține câmpurile, tipurile de date ale acestora, restricțiile și referințele către alte entități. În cadrul aplicației noastre, am definit următoarele modele:

- **User:** reține pentru fiecare utilizator numele, prenumele, adresa de email, numele de utilizator, parola și poza de profil. De asemenea, adaugă și restricții pentru a asigura unicitatea numelui de utilizator și a adresei de email, tipuri de date pentru fiecare câmp, necesitatea completării câmpurilor și lungimea minimă și maximă a câmpurilor.
- **Video:** reține pentru videoclip informații precum url-ul unde este stocat, titlul, descrierea, id-ul autorului, textul din videoclip și calea către subtitrări în formatul .vtt, o listă cu id-urile utilizatorilor care au dat like/dislike, topicul videoclip-ului

și data. Restricțiile impuse sunt similare cu cele de la modelul User, au grijă ca fiecare videoclip să aibă obligatoriu metadatele necesare și să aibă un autor valid. De asemenea, sunt menționate și tipurile de date ale câmpurilor.

- **Comment:** reține pentru fiecare comentariu textul, id-ul autorului, id-ul videoclipului, dacă este șters sau nu (*soft-delete/hard-delete*) pentru eficientizarea procesului de ștergere a comentariilor, și id-ul comentariului la care răspunde pentru a putea crea ierarhia comentariilor. Schema modelului impune restricția ca id-ul autorului și id-ul videoclip-ului să fie obligatorii, iar textul să fie completat.

## Controllers

Fiecare rută menționată anterior are asociată un controller care procesează cererea primită, apelează serviciile necesare și returnează răspunsul.

Menționez aici câteva dintre funcțiile mai importante:

- **register/login:** funcții care se ocupă de înregistrarea și logarea utilizatorilor, creează token-ul JWT folosind secretul stocat pe server și id-ul utilizatorului, verifică existența utilizatorului în baza de date și returnează token-ul JWT.
- **getUserByUsername/updateUserProfile:** funcții folosite pentru a obține informații despre utilizator, respectiv pentru a actualiza informațiile utilizatorului. Se observă aici cum funcția de actualizare a profilului folosește atât middleware-ul de autentificare pentru a se asigura că doar utilizatorul autentificat poate actualiza profilul, cât și middleware-ul de încărcare a fișierelor pentru a actualiza poza de profil. Ambele funcții accesează baza de date pentru a obține sau actualiza informațiile și returnează răspunsul.
- **addComment/updateComment/deleteComment/getCommentsForVideo:** după cum sugerează și numele, sunt funcții care permit adăugarea, actualizarea, ștergerea și obținerea comentariilor pentru un videoclip. Funcțiile de adăugare și actualizare a comentariilor folosesc middleware-ul de autentificare pentru a se asigura că doar utilizatorii autentificați pot executa aceste operații. Întâlnim aici și funcția *populate* din Mongoose care ne permite să expandăm referințele din baza de date, în cazul nostru, să obținem informații despre autorul comentariului.
- **uploadVideo/updateVideo/deleteVideo:** funcții care permit încărcarea videoclipurilor, actualizarea și ștergerea acestora. Fiecare funcție necesită middleware-ul de autentificare pentru a asociera unui videoclip cu autorul său, iar funcțiile de actualizare și ștergere a videoclip-urilor permit doar autorului videoclip-ului să execute aceste operații. Serviciul de încărcare a video-urilor apelează la rândul lui

cele două servere de Flask pentru extragerea subtitrărilor și clasificarea videoclipului.

- **getAllVideos/getVideosByUser/getVideoById**: funcții care permit accesarea videoclipurilor din baza de date în diferite moduri: toate videoclipurile, videoclipurile unui utilizator sau un videoclip anume după id-ul său. Funcțiile caută la rândul lor în baza de date după criteriile specificate și returnează răspunsul.
- **likeVideo/dislikeVideo/getLikes/getDislikes**: funcții care permit interacțiunea cu un videoclip, adăugând sau ștergând id-ul utilizatorului care a dat like/dislike din lista de referințe a videoclipului. De asemenea, sunt tratate aici și cazurile speciale în care utilizatorul dă dislike după ce a dat like sau invers și cazul în care utilizatorul retrage aprecierea.
- **searchVideos**: funcție care primește din cerere cuvintele cheie (titlu, descriere, subtitrări, topic), accesează baza de date Elasticsearch pentru a căuta videoclipurile care conțin acele cuvinte și returnează o listă cu id-urile videoclipurilor găsite. De menționat că am ales un threshold de 1.0 pentru a filtra rezultatele mai slabe și am setat parametrul *fuzziness* pe *AUTO* pentru a permite căutări aproximative.

## Utils

Funcțiile utilitare nu depind de o anumită entitate și sunt folosite pentru a facilita procesarea cererilor. În cadrul aplicației noastre, am definit următoarele funcții utilitare:

- **getTranscription**: înainte de a trimite cererea către serverul de Flask pentru recunoașterea vorbirii, e nevoie să extragem întâi secvența audio din videoclip deoarece modelul lucrează cu fișiere audio. Folosesc pentru acest lucru librăria *ffmpeg-extract-audio* care permite procesarea fișierelor video și extragerea secvenței audio. De asemenea, secvența audio fiind destul de mare, creez un flux de date folosind *fs.createReadStream* și atașez acest flux la cererea către serverul de Flask.
- **getTopic**: după ce am obținut subtitrările videoclipului, folosesc un serviciu extern pentru a clasifica fiecare videoclip în funcție de metadatele sale și a subtitrărilor. Astfel, creez o cerere către serverul de Flask pentru clasificare și returnez topicul videoclipului.

## Config

Configurările sunt folosite pentru a inițializa comunicarea cu alte servicii precum baza de date Elasticsearch. Folosesc două fișiere de configurare: *elasticsearchClient.ts* și *elasticsearchSetup.ts*. Primul creează un client Elasticsearch folosind librăria *@elastic/elasticsearch* și exportă clientul pentru a fi folosit în servicii, iar al doilea creează un index specific

Elasticsearch pentru a stoca informații necesare căutării videoclipurilor. Deoarece aplicația este un prototip, folosesc o singură replică pentru index și un singur shard.

## Expunerea datelor

Datele multimedia precum: videoclip-urile, subtitrările și pozele de profil sunt stocate pe mașina serverului, în directoare separate: */uploads*, */captions* și */profile-pictures*. Aceste directoare sunt expuse către client folosind *express.static* și pot fi accesate la link-ul unde rulează serverul concatenat cu numele fișierului stocat în baza de date.

```
app.use('/uploads', express.static('uploads'));
app.use('/captions', express.static('captions'));
app.use('/profile-pictures', express.static('profile-pictures'));
```

## Criptarea parolelor

Serverul folosește algoritmul de criptare *bcrypt* pentru a stoca parolele utilizatorilor în baza de date. Algoritmul de criptare are la bază următoarele concepte:

- **Salting:** librăria *bcrypt* generează automat un salt (o valoare aleatoare) pentru fiecare parolă, pentru a preveni atacurile de tip *rainbow table*. Salt-ul este adăugat la parolă înainte de a fi criptată. Astfel, chiar dacă două parole sunt identice, salt-ul diferit va genera hash-uri diferite.
- **Cost Factor:** procesul de criptare implică aplicarea algoritmului de hashing de un anumit număr de ori, numit *cost factor*. Acest proces încetinește semnificativ viteza de criptare, făcând astfel atacurile cu forță brută ineficiente, în special pentru atacurile la scară largă.

## Algoritm de criptare bcrypt

Utilizarea algoritmului de criptare *bcrypt*:

- **Input:** *bcrypt* primește ca input parola utilizatorului, salt-ul generat și parametrul de cost. Valoarea implicită a costului este 10, dar poate fi modificată în funcție de necesitatea aplicației și de puterea de calcul a serverului.
- **Processing:** algoritmul de criptare folosește Eksblowfish, un algoritm de hashing variație a Blowfish, dar modificat să fie mai lent și să includă salt-ul și parola.
- **Output:** rezultatul este un hash de 60 de caractere care conține informații despre versiunea de *bcrypt*, cost parameter, salt și hash-ul parolei.

Algoritmul de criptare *bcrypt* produce o ieșire de forma:

```
$2<a/b/x/y>$[cost]$[22 character salt][31 character hash]
```

Exemplu de criptare a parolei "abc123xyz"<sup>1</sup>:

[illegible]

## Variabile de mediu

Serverul folosește variabile de mediu stocate în fișierul `.env` pentru a putea lăsa la îndemâna utilizatorului anumite configurații precum portul pe care rulează serverul, secretul folosit pentru generarea token-ului JWT și link-urile către serverele de Flask. Variabilele de mediu sunt citite folosind librăria `dotenv` și sunt accesate în cod folosind `process.env`.

În cadrul aplicației noastre, am folosit următoarele variabile de mediu:

- **PORT**: portul pe care rulează serverul, implicit 5000.
- **JWT\_SECRET**: secretul folosit pentru generarea token-ului JWT.
- **BACKEND\_ASR\_URL**: link-ul către serverul de Flask pentru recunoașterea vorbirii.
- **BACKEND\_TOPIC\_URL**: link-ul către serverul de Flask pentru clasificarea videoclip-urilor.
- **BASE\_URL**: link-ul către serverul Express, folosit pentru a construi link-urile
- **MONGO\_URI**: link-ul către baza de date MongoDB, conține și credențialele de acces.

## Pornirea serverului

Am configurat câmpul *scripts* din fișierul *package.json* pentru a customiza comenzile de pornire a serverului. Astfel, pentru modul de dezvoltare, am definit comanda *dev* care pornește serverul folosind *nodemon src/server.ts* și repornește serverul automat la modificarea fișierelor. Pentru modul de producție, am definit comanda *start* care pornește serverul folosind *node dist/server.js*.

Există două directoare `/src` și `/dist` pentru a separa codul sursă de codul compilat. Astfel, pentru eficientizarea codului în producție, codul este compilat o singură dată și rulat folosind fișierul `dist/server.js`. Pentru a face acest lucru, am folosit librăria `tsc` care dă `build` și rulează comanda `npx tsc`.

---

<sup>1</sup>Exemplul este preluat de pe Wikipedia la adresa: <https://en.wikipedia.org/wiki/Bcrypt>



### 3.3.2 Flask Server - Recunoașterea vorbirii

Deși este un server de Flask, structura este similară cu cea a serverului Express în sensul că este împărțit în app, rute, controlere, config și utils.

#### Routes

Sintaxa specifică Flask-ului pentru definirea rutelor constă în a folosi decoratorul `@app.route` pentru a asocia o funcție cu o cale specifică și pentru a specifica metoda HTTP folosită.

- **/transcribe**: ruta care primește cererea de la serverul Express, extrage secvența audio, verifică ca extensia fișierului să fie validă, creează un fișier temporar pentru a stoca secvența audio, apelează funcția de recunoaștere a vorbirii și returnează textul transcris.
- **/vtt/<filename>**: ruta care returnează subtitrările în formatul .vtt pentru videoclip. Funcția primește numele fișierului și returnează subtitrările stocate în acel fișier.
- **/vtt/delete/<filename>**: ruta care șterge subtitrările pentru videoclip. Această funcție este apelată după ce subtitrările au fost livrate cu succes pentru a elibera spațiul de stocare.

#### Audio

Înainte de a explica cum funcționează recunoașterea vorbirii, vreau să explic ce este un fișier audio, cum este reprezentat și cum este procesat.

Definim un convertor analog-digital ca fiind un dispozitiv care transformă semnalul recepționat de la microfon (variații în presiunea aerului/unde sonore) într-un semnal digital. Practic, convertește semnalul analogic continuu într-un semnal digital discret care poate fi stocat și procesat de un calculator. Acest proces implică conceptul de cuantizare și introduce o eroare de cuantizare care depinde de rezoluția convertorului. În acest sens, convertorul analog-digital convertește periodic semnalul analogic, eșantionând semnalul la intervale de timp egale.

Perioada de timp dintre două eșantioane consecutive este cunoscută sub numele de **frecvență de eșantionare** și este măsurată în Hz. Cu cât frecvența de eșantionare este mai mare, cu atât semnalul digital este mai apropiat de semnalul analogic original. Această frecvență măsoară numărul de sample-uri pe secundă.

Conceptul de **rezoluție** indică numărul de valori discrete distincte pe care le poate reprezenta un convertor analog-digital. Cu cât rezoluția este mai mare, cu atât semnalul digital este mai apropiat de semnalul analogic original.

De exemplu, un convertor analog-digital cu o rezoluție de 8 biți poate encoda un semnal analogic în  $2^8 = 256$  de valori discrete. Figura de mai jos ilustrează procesul de conversie analog-digital și digital-analog. 3.14

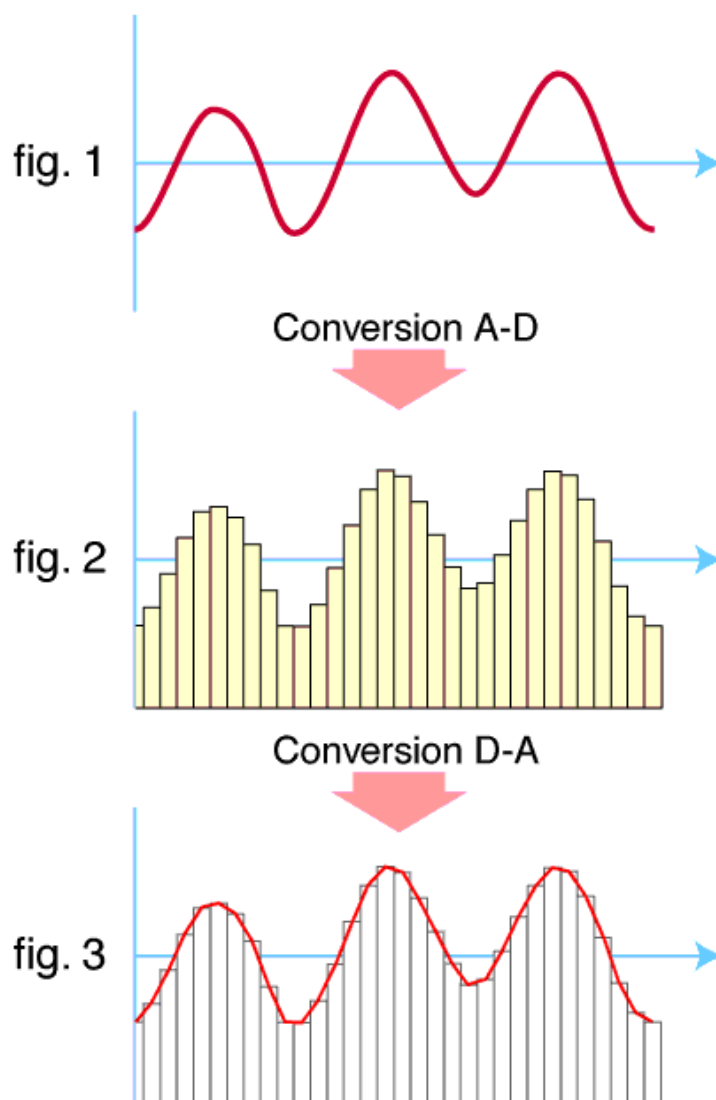


Figura 3.14: Procesul de conversie analog-digital și digital-analog<sup>1</sup>

Acum că am definit cum sunt interpretate semnalele audio, putem să trecem la procesarea acestora. Definim ce semnifică valorile obținute:

- **Amplitudinea:** reprezintă presiunea undelor sonore măsurată la un moment de timp. Cu cât amplitudinea este mai mare, cu atât sunetul este mai puternic.
- **Amplitudinea zero:** valoarea 0 reprezintă linia de bază a semnalului audio, punctul principal de referință după care se măsoară amplitudinea.
- **Valori pozitive și negative:** valorile sunt reprezentate în formatul PCM (Pulse Code Modulation), unde valorile pozitive sunt reprezentate de valori între 0 și 1, iar valorile negative sunt reprezentate de valori între 0 și -1.

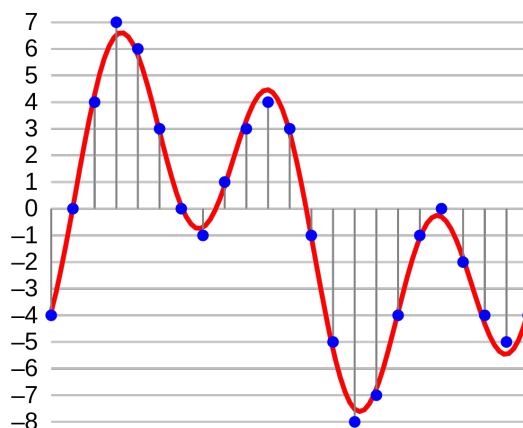


Figura 3.15: Reprezentarea semnalului audio în format PCM<sup>2</sup>

De aici încolo vom lucra cu semnalul digital în Python folosind librăria *soundfile*, *librosa* și *numpy* pentru a procesa secvențele audio.

## Controllers

Pentru a menține logica din rute cât mai simplă, am creat un controller care se ocupă de procesarea secvențelor audio primite de la serverul Express.

## Procesarea secvențelor audio

Funcția `speech_to_text` primește ca parametru calea către fișierul audio salvat temporar, îl deschide folosind *soundfile* și îl transformă într-un șir de float-uri. Sunt două tipuri de fișiere audio: **mono** și **stereo**, iar funcția trebuie să facă distincția între cele două tipuri.

Formatul *mono* are un singur canal audio, iar formatul *stereo* are două canale audio, fiind conceput pentru a reda sunetul într-un mod mai realist. Deoarece modelul de recunoaștere a vorbirii acceptă doar fișiere audio mono, am făcut media celor două canale pentru a obține un singur canal audio.

De asemenea, modelul a fost antrenat pe fișiere audio cu o frecvență de eșantionare de 16kHz, așa că în cazul în care frecvența fișierului audio este diferită (de exemplu, 44.1kHz), trebuie să o convertim la 16kHz. Acest lucru se face folosind funcția *resample* din librăria *resampy*.

Un aspect important, după cum am menționat și la seturile de date folosite, este că modelul a fost antrenat pe fișiere audio cu o durată de aproximativ 10 secunde. O secvență audio prea mare oricum nu ar încăpea în memoria serverului, așa că am ales

<sup>1</sup>Imagine preluată de pe Wikipedia la adresa: [https://en.wikipedia.org/wiki/Analog-to-digital\\_converter](https://en.wikipedia.org/wiki/Analog-to-digital_converter)

<sup>2</sup>Imagine preluată de pe Wikipedia la adresa: [https://en.wikipedia.org/wiki/Audio\\_bit\\_depth](https://en.wikipedia.org/wiki/Audio_bit_depth)

să sparg secvența audio în segmente de câte 30 de secunde folosind funcția *stream* din librăria *librosa*.

Un alt aspect de luat în calcul aici, chiar dacă modelul ar putea procesa secvențe mai lungi, mai intervine și timeout-ul socket-ului care ar putea expira în cazul unor secvențe prea mari. Deși în cazul spargerii secvenței audio în segmente de câte 30 de secunde, se pot pierde informații de la începutul și finalul fiecărei bucăți, am ales să sacrific aceste informații pentru a asigura procesarea cu succes a secvențelor audio. Exceptând aceste cazuri, bucățile pot fi tratate independent și deci procesate în paralel, îmbunătățind semnificativ timpul de răspuns.

## Subtitrări

Funcția **make\_subtitles** primește ca parametru textul transcris din fiecare bloc de 30 de secunde și creează subtitrările în formatul *.vtt*. În cadrul unui bloc, modelul de recunoaștere a vorbirii folosește următoarele etape:

- **Processing:** modelul primește secvența audio, și folosește procesorul specific de pe Huggingface pentru a converti secvența audio în tensori. Vom numi acești tensori *input\_values*.
- **Prediction:** modelul primește *input\_values* și returnează *logits*, tot sub forma de tensori reprezentând probabilitățile pentru fiecare literă din vocabular. Sunt calculate apoi *predicted\_ids* fiind literele cu cele mai mari probabilități.
- **Decoding:** folosind *logits*, modelul folosește algoritmul de decodare pentru a converti tensorii în text și obținem transcription.

Astfel, obținem pentru fiecare bloc *transcription*, *input\_values* și *predicted\_ids*.

Pentru a crea subtitrările, folosim următoarele etape:

- **ids\_w\_time:** cunoscând durata întregului bloc (30 de secunde) și numărul de token-uri prezis de model, putem estima momentul de timp al fiecărui token prin împărțirea indexului token-ului la numărul total de token-uri și înmulțirea cu durata blocului. Putem elimina token-urile pentru [PAD].
- **split\_ids\_w\_time:** folosind token-ul de delimitare între cuvinte, putem crea câte un grup de token-uri între fiecare delimitator. Practic, obținem cuvintele propriu-zise, dar acum știm și momentul de timp.
- **word\_timestamps:** pentru fiecare cuvânt, calculăm momentul de timp de început și de sfârșit ca fiind minimul, respectiv maximul dintre toate momentele de timp ale token-urilor din acel cuvânt.

- **group\_timestamps**: putem grupa acum cuvintele în blocuri de 7 cuvinte, pentru a contura subtitrările. Începutul și sfârșitul fiecărui bloc de cuvinte este dat de momentul de timp al primului și ultimului cuvânt din bloc.

```

predicted_ids: [6, 6, 0, 8, 0, 4, 4, 0, 17, 0, 5, 5, 0, 7, 0, 6,
                \_____/                \_____/
                  to                    |                    meet
0, 0, 4, 4, 0, 0, 18, 0, 0, 7, 0, 0, 12, 12, 4, 4, 6, 0, 0, 0, 8,
                \_____/                \_____/
                  |                    |                    to
0, 4, 0, 0, 20, 0, 0, 10, 0, 0, 9, 0, 14, 0, 4, 4, 4, 5, 5, 7, 0,
                \_____/                \_____/
                  |                    |                    each
19, 11, 0, 4, 4, 0, 0, 8, 8, 0, 6, 11, 11, 0, 5, 13, 0, 4, 4, 4]
_____/                \_____/                |
                  |                    other

```

```

words: ['to', 'meet', 'was', 'to', 'find', 'each', 'other']

```

ids for letters and delimiter, but without pad

```

[6, 6, 8, 4, 4, 17, 5, 5, 7, 6, 4, 4, 18, 7, 12, 12, 4, 4, 6, 8,
 \_____/                \_____/                \_____/                \_____/
   to                    |                    meet                    |                    was                    |                    to
4, 20, 10, 9, 14, 4, 4, 4, 5, 5, 7, 19, 11, 4, 4, 8, 8, 6, 11, 11,
| \_____/                |                \_____/                |                \_____/
5, 13, 4, 4, 4]
_____/                |

```

```

split_ids_w_time

```

```

'to' -> [(0.54, 6), (0.56, 6), (0.64, 8)]
'meet' -> [(0.74, 17), (0.78, 5), (0.80, 5), (0.84, 7), (0.88, 6)]
'was' -> [(1.48, 18), (1.54, 7), (1.60, 12), (1.62, 12)]
'to' -> [(1.69, 6), (1.77, 8)]
'find' -> [(1.87, 20), (2.01, 10), (2.07, 9), (2.11, 14)]
'each' -> [(2.21, 5), (2.23, 5), (2.25, 7), (2.29, 19), (2.31, 11)]
'other' -> [(2.43, 8), (2.45, 8), (2.49, 6), (2.51, 11), (2.53, 11),

```

```
(2.57, 5), (2.59, 13)]
```

```
word_timestamps
```

```
'to' -> 0.54 - 0.64  
'meet' -> 0.74 - 0.88  
'was' -> 1.48 - 1.62  
'to' -> 1.69 - 1.77  
'find' -> 1.87 - 2.11  
'each' -> 2.21 - 2.31  
'other' -> 2.43 - 2.59
```

```
group_timestamps
```

```
'to meet was to find each other' -> 0.54 - 2.59
```

## Formatarea subtitrărilor

Funcția **save\_subtitles** primește ca parametru subtitrările și creează un fișier *.vtt*, respectând formatul specificat. Fișierele în formatul *.vtt* încep cu antetul *WEBVTT*, iar pentru fiecare grup de subtitrări se specifică momentul de timp de început și de sfârșit, urmat de textul subtitrării.

Mai jos este un exemplu de subtitrare în formatul *.vtt*:

```
WEBVTT
```

```
00:00:04.162 --> 00:00:06.424  
sometimes math and physics conspire in ways
```

```
00:00:06.524 --> 00:00:07.665  
that just feel too good to be
```

```
00:00:07.745 --> 00:00:09.706  
true let's play a strange sort of
```

Inspirat din issue-ul de pe GitHub: <https://github.com/huggingface/transformers/issues/11307>

## Config

Configurările sunt folosite pentru a specifica serverului cum să proceseze cererile primite.

- **ASR\_REPO**: numele modelului de recunoaștere a vorbirii folosit pentru a transcrie secvențele audio.
- **BOOSTED\_LM**: dacă este setat pe *true*, folosește varianta îmbunătățită cu un n-gram language model pentru recunoașterea vorbirii.
- **SAMPLE\_RATE**: frecvența de eșantionare a fișierelor audio, setată pe 16kHz pentru a respecta cerințele modelului.
- **BLOCK\_SIZE**: lungimea blocurilor folosite în fluxul de date, setată pe 30 de secunde.
- **SPELL\_CHECK**: dacă este setat pe *true*, folosește un model de corectare a cuvintelor pentru a îmbunătăți rezultatele.

### 3.3.3 Flask Server - Clasificarea videoclipurilor

Serverul de Flask pentru clasificarea videoclipurilor este similar cu cel pentru recunoașterea vorbirii, dar are o structură mai simplă, nefiind necesare mai multe etape de procesare.

#### Routes

Astfel, serverul definește o singură rută */topic* care primește cereri de tipul *POST* și returnează topicul textului primit. Deoarece secvența de intrare a modelului *BERT* este limitată la 512 token-uri, am ales să împart textul în blocuri de câte 512 token-uri și fiecare bloc este trimis la model pentru a obține topicul. Pentru a obține topicul final, se contorizează frecvența apariției fiecărui topic și se returnează cel majoritar.

#### Configurări

Serverul creează un *pipeline* specific Huggingface înainte de a primi cereri, pentru a reduce timpul de răspuns. Acest *pipeline* conține modelul preantrenat și fine-tunat pe setul de date *BBC News*. Mai multe detalii se regăsesc în secțiunea 2.2.

## 3.4 Baze de date

Pentru stocarea datelor, am ales să folosesc două baze de date: MongoDB, datorită structurii sale flexibile care permite dezvoltarea rapidă a aplicațiilor și Elasticsearch, pentru căutarea eficientă în metadatele videoclipurilor.

### 3.4.1 MongoDB

#### Arhitectura bazei de date

MongoDB este o bază de date de tip **NoSQL** care stochează datele sub forma de documente în formatul *JSON*. Documentele sunt grupate în colecții, iar colecțiile sunt apoi grupate în baze de date.

Intuitiv, un document este echivalentul unui rând dintr-o bază de date relațională, dar cu o structură similară unui obiect JSON. Acesta permite stocarea datelor fără a respecta un model de date fix, adică pot fi adăugate câmpuri noi, chiar și alte documente, diferite de celelalte documente din aceeași colecție.

#### Structura datelor

Fiecare document conține, pe lângă datele propriu-zise, câmpuri speciale precum:

- **\_\_id**: un *ObjectId* adăugat automat de MongoDB pentru a identifica unic documentul.
- **\_\_v**: un câmp special care indică versiunea documentului în cazul actualizării. Acest câmp a fost conceput pentru a preveni conflictele de actualizare în cazul cererilor concurente.
- **createdAt** și **updatedAt**: câmpuri speciale de tipul *ISODate* care indică momentul de creare și actualizare pentru fiecare document.

#### Colecții

În cadrul aplicației noastre, am definit 3 colecții:

- **users**: conține informații despre utilizatori precum: numele și prenumele, adresa de email, numele de utilizator, parola criptată și url-ul către poza de profil.
- **videos**: conține informații despre videoclipuri precum: titlul, descrierea, topicul, id-ul autorului (referință către colecția *users*), url-ul către videoclip, url-ul către subtitrări și două liste separate pentru id-urile utilizatorilor care au dat like și dislike.
- **comments**: conține informații despre comentarii precum: textul comentariului, id-ul autorului (referință către colecția *users*), id-ul videoclipului (referință către colecția *videos*), *isDeleted* folosit la *soft-delete* și *hard-delete*, *parentId* în cazul în care comentariul este un răspuns la alt comentariu.



## Interacțiunea cu baza de date

Interacțiunea cu baza de date are loc cu ajutorul librăriei *mongoose* care definește modelele în serverul Express și execută operații de tip CRUD (Create, Read, Update, Delete) în baza de date. *mongoose* este un ODM (Object Data Modeling) care oferă un nivel de abstractizare peste MongoDB și permite definirea de scheme pentru documente, validarea datelor și crearea de relații între colecții.

Un alt aspect important pus la dispoziție de **MongoDB** îl reprezintă capacitatea de a crea replici la nivel de colecție, numite *indexes*. Deși aplicația noastră este un prototip, nu are nevoie de scalările orizontale, dar în cazul unui volum mare de date, replicile facilitează operațiile de citire și scriere, îmbunătățind timpul de răspuns.

### 3.4.2 Cron Jobs

#### Definiție și utilitate

Un **cron job** este un proces care rulează automat la un momente de tip prestabilite. În cadrul proiectului, un *cron job* își găsește utilitatea în procesul de curățare al arborelui de comentarii pentru fiecare videoclip. Problema constă în găsirea unei soluții eficiente care șterge comentariile marcate drept *soft-delete* cu subarborele de răspunsuri gol.

Figura de mai jos ilustrează acest concept:

```
Cool video!
|__ Indeed it is!
|__ |__ [deleted] (1)
|__ [deleted] (2)
|__ |__ Waiting for the next one!
[deleted] (3)
Awesome!
|__ [deleted] (4)
|__ |__ Yeah, I agree!
|__ [deleted] (5)
|__ |__ [deleted] (6)
```

În exemplul de mai sus, comentariile cu textul [deleted] sunt marcate drept *soft-delete*, semnificând că deși au fost șterse de utilizator, ele încă există în baza de date.

Problema apare atunci când un comentariu este marcat drept *soft-delete*, dar nu știm, fără o parcurgere în prealabil, dacă poate fi șters sau nu.

O primă idee ar fi ca de fiecare dată când un comentariu este șters, să facem o cerere pentru întreg arborele de comentarii și să verificăm dacă fiecare nod are copii care pot fi la rândul lor șterși. Această metodă este inefficientă, deoarece necesită o cantitate mare de resurse pentru a parcurge întreg arborele de comentarii la fiecare actualizare.

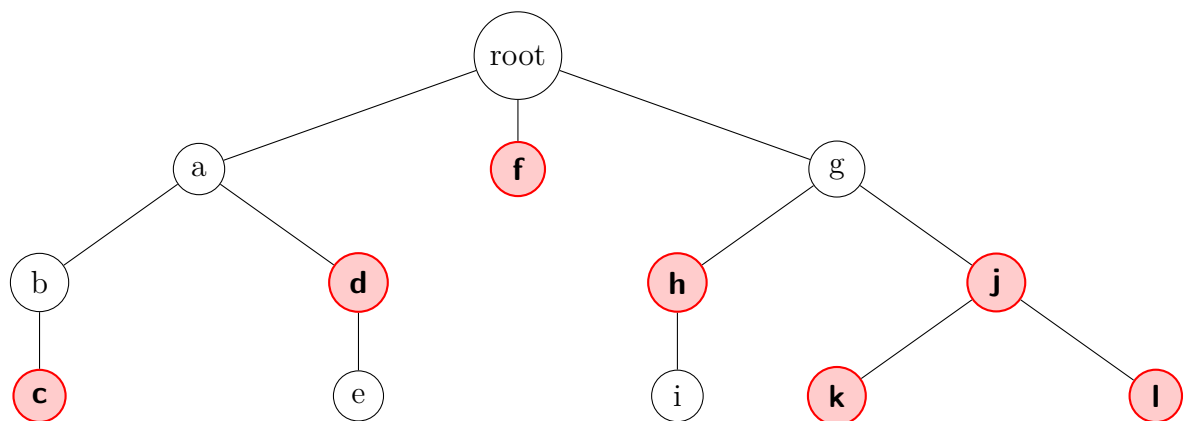
O altă soluție ar fi să păstrăm comentariile în starea *soft-delete* și să rulăm un *cron job* la intervale regulate de timp care să curețe arborele de comentarii.

## Algoritm

Algoritmul de curățare a arborelui de comentarii este următorul:

- **Inițializare:** creează conexiunea cu baza de date, obține toate comentariile și creează structura arborescentă menționată anterior.
- **Parcursere:** parcurge arborele în adâncime folosind algoritmul DFS (Depth First Search) și verifică recursiv folosind conceptul de programare dinamică, dacă un nod poate fi șters sau nu. La fiecare pas din recursie se verifică dacă nodul curent este marcat drept *soft-delete* și dacă toți subarborii copiilor săi sunt de asemenea marcați drept *soft-delete*. În caz afirmativ, nodul curent poate fi șters și este adăugat într-o listă de noduri *deletable*. Altfel, nodul nu poate fi șters și implicit toate nodurile de pe drumul până la rădăcină nu pot fi șterse.
- **Ștergere:** după ce arborele a fost parcurs și a fost creată lista *deletable*, se apelează iar clientul MongoDB pentru a șterge toate nodurile din listă.

Exemplul de mai jos ilustrează algoritmul de curățare a arborelui de comentarii:



În arborele prezentat, nodurile colorate cu roșu reprezintă comentariile *soft-delete*. Algoritmul va adăuga nodurile *c*, *f*, *j*, *k* și *l* în lista *deletable*.

Chiar dacă nodurile *d* și *h* sunt marcate drept *soft-delete*, ele nu pot fi șterse permanent (*hard-delete*) din cauza subarborilor încă activi.

### 3.4.3 Elasticsearch

#### Concept

Elasticsearch este un motor de căutare ce organizează datele în documente în formatul *JSON* care sunt apoi grupate în indici. Indicii sunt echivalentul bazelor de date din MongoDB, iar documentele sunt echivalentul rândurilor din tabelele relaționale.

Concepte cheie în Elasticsearch:

- **Document:** unitatea de bază folosită de Elasticsearch în formatul *JSON* (considerat formatul standard pentru transferul de date)
- **Indices:** o colecție de documente cu trăsături similare formează un indice și reprezintă cel mai înalt nivel de organizare a datelor.
- **Inverted Index:** reprezintă mecanismul care stă la baza motoarelor de căutare și constă într-o structură de date (similară cu un hashmap) care mapează cuvintele (sau mai bine zis, termenii) la locațiile unde apar în documente. Astfel, fiecare document este împărțit în termeni individuali care sunt folosiți la căutare.

Figura 3.16 ilustrează conceptul *Inverted Index*. [6]

#### Inserare

În cazul aplicației noastre, am definit un singur indice *videos* care conține metadatele videoclip-urilor: titlul, descrierea, topicul și subtitrările. De fiecare dată când un videoclip este încărcat în aplicație, inserarea se face atât în baza de date MongoDB, cât și în Elasticsearch.

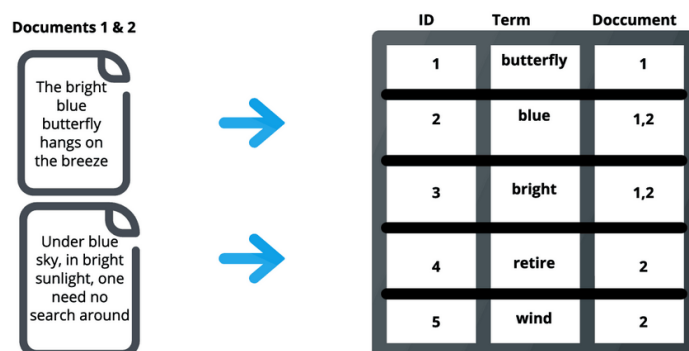


Figura 3.16: Inverted Index<sup>3</sup>

<sup>3</sup>Imagine preluată de pe site-ul knowi.io la adresa: <https://www.knowi.com/blog/what-is-elastic-search/>

## Căutare

Interogarea în Elasticsearch permite specificarea criteriului de căutare precum: full-text, fuzzy, prefix, expresii regulate etc. În cazul aplicației noastre, am ales să folosesc *fuzziness AUTO* pentru a permite căutarea cuvintelor similare cu cele din cerere.

## 3.5 Deployment

### 3.5.1 Docker

#### Concept

Docker este o platformă care permite dezvoltatorilor să izoleze logica, dependențele și mediul de lucru în containere. Spre deosebire de mașinile virtuale, Docker nu virtualizează întregul sistem de operare, ci combină aplicația și dependențele sale cu librăriile oferite de sistemul de operare.

În contextul mașinilor virtuale (VM), alocarea de resurse este dirijată de un *hypervisor* care gestionează resursele fizice ale sistemului. În mod asemănător, Docker folosește un sistem de gestiune al resurselor mult mai eficient din punct de vedere al memoriei și timpului de rulare.

Avantajele folosirii containerelor Docker sunt:

- **memorie mică:** un container Docker nu conține întregul sistem de operare, ci doar procesele și dependențele necesare pentru aplicație, dimensiunea acestuia fiind de ordinul MB.
- **portabilitate:** rezolvă problema *it works on my machine* datorită izolării mediului de lucru cu ajutorul imaginilor și containerelor.
- **utilizare eficientă a resurselor:** permite gestionarea eficientă a resurselor sistemului.

#### Dockerfile și Docker image

Imaginile Docker reprezintă unitatea de bază a containerelor și conțin tot ce este necesar (codul sursă, dependențele, tool-urile, librăriile, variabile de mediu etc.) pentru a rula o aplicație. Imaginile sunt structurate în nivele de abstractizare construite pe baza unui fișier numit *Dockerfile*.

Un *Dockerfile* enumeră instrucțiunile necesare pentru construirea imaginii. De obicei, se pleacă de la o imagine de bază (de exemplu, *python:3.8-slim*), după care se instalează dependențele necesare și se copiază codul sursă în container. În ultima parte a fișierului, se specifică comanda care pornește aplicația.

Odată ce *Dockerfile*-ul este definit, se pot construi imagini Docker folosind comanda *docker build*, eventual cu argumente suplimentare în linia de comandă. (de exemplu, menționarea portului pe care rulează aplicația, versiunea imaginii etc.)

## Docker container

Un container de Docker folosește imaginea construită anterior și creează instanțe ale acesteia. În timp ce o imagine permite doar citirea, un container permite dezvoltatorului interacțiunea cu acesta, fie pentru a rula comenzi, fie pentru a monitoriza starea aplicației.

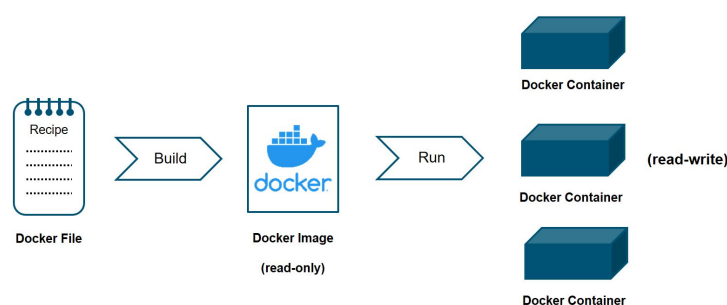


Figura 3.17: Docker<sup>4</sup>

## Docker Compose

Pe parcursul dezvoltării aplicației, apar din ce în ce mai multe servicii care trebuie gestionate separat și care depind unele de altele. Docker Compose este un tool care înglobează toate comenzile necesare pentru pornirea și oprirea serviciilor, precum și pentru construirea imaginilor și a containerelor.

Docker Compose folosește un fișier numit *docker-compose.yml* care conține toate detaliile necesare pentru fiecare serviciu, dintre care amintim: imaginea folosită, numele containerului, porturile expuse, variabilele de mediu, volumele, dependențele între alte servicii, rețelele folosite etc.

## Volume și rețele

În cazul în care aplicația are nevoie de o stocare persistentă, de pildă pentru bazele de date, se pot folosi volume Docker. Un volum Docker este, de fapt, un director din sistemul gazdă care este montat în container al cărui conținut este persistent chiar și după oprirea containerului.

De asemenea, Docker oferă și posibilitatea de a crea rețele virtuale prin intermediul cărora serviciile pot comunica între ele. Implicit, Docker oricum atribuia fiecărui container o adresă IP, dar în cazul pornirilor succesive ale aceluiași container, adresa IP se schimbă

---

<sup>4</sup>Imagine preluată de pe site-ul suresoft la adresa: <https://suresoft.gitlab-pages.rz.tu-bs.de/workshop-website/continuous-integration/containers.html>

și nu mai poate fi accesată. Folosirea rețelelor Docker permite atribuirea manuală a IP-urilor.

### 3.5.2 Dockerizarea aplicației

Aplicația beneficiază de avantajele oferite de Docker și Docker Compose pentru a simplifica procesul de dezvoltare și de deployment. În cadrul proiectului, am definit următoarele servicii:

- **MongoDB Container:** rulează o instanță de MongoDB pentru stocarea datelor, folosind un volum Docker pentru a asigura persistența datelor și făcând o corespondență între portul gazdă și portul containerului.
- **Elasticsearch Container:** rulează o instanță de Elasticsearch pentru căutarea eficientă în metadatele videoclip-urilor, folosind de asemenea un volum Docker.
- **Flask Server - Recunoașterea Vorbirii:** rulează serverul Flask pentru recunoașterea vorbirii, folosind portul 5001. Deoarece serverul comunică cu MongoDB și Elasticsearch, cele două servicii sunt specificate ca dependențe.
- **Flask Server - Clasificarea Videoclipurilor:** rulează serverul Flask pentru clasificarea videoclipurilor, folosind portul 5003. Nu are dependențe.

Pentru serverele de React și Node.js am ales să nu folosesc containere Docker astfel încât să pot beneficia de o dezvoltare rapidă și de actualizare în timp real a codului sursă.

# Capitolul 4

## Concluzie

### 4.1 Concluzie

Prin realizarea acestei lucrări am prezentat conceptul de recunoaștere a vorbirii dintr-un videoclip integrându-l într-o aplicație web relevantă. Deși scopul proiectului a fost o aplicație web, librăria Material UI oferă posibilitatea de a fi folosită și pe dispozitivele mobile.

Inițial, mi-am îndreptat atenția către modelul de recunoaștere a vorbirii și generarea subtitrărilor, dar consider că integrarea acestuia într-un context concret ilustrează mai bine utilitatea și importanța acestuia. De aceea, mi-am structurat lucrarea de licență în două părți: partea de **Inteligență Artificială**, în care am detaliat arhitectura modelului *wav2vec 2.0*, cum am ales setul de date și cum am antrenat modelul, și arhitectura modelului *BERT* pentru clasificarea videoclip-urilor, explicată în aceeași manieră, folosită pentru a îmbunătăți algoritmul de căutare, precum și partea de **Inginerie Software**, în care am prezentat arhitectura aplicației web, tehnologiile folosite, structura frontend-ului, backend-ului, bazelor de date, dar și a serviciilor de deployment și a serviciilor utilitare (Cron Jobs).

Așa cum am menționat în secțiunea *Inginerie Software*, aplicația oferă utilizatorilor funcționalități precum: autentificare, CRUD pe informațiile proprii, dar și pe videoclip-uri și interacțiunea cu acestea, căutarea videoclip-urilor după cuvinte cheie și suport pentru recunoașterea vorbirii, crearea subtitrărilor și clasificarea videoclip-urilor.

În procesul de dezvoltare al aplicației, am gândit întâi funcționalitățile pe care le urmăresc și apoi am ales tehnologiile potrivite pentru a le implementa. Acest context mi-a oferit oportunitatea de a învăța aceste tehnologii și de a le aprofunda în lucrarea de licență.

Am ales deci să utilizez stack-ul **MERN** (MongoDB, Express.js, React.js, Node.js), folosind limbajul TypeScript la care am adăugat și două servere de Flask în Python. Am ales să fac partea de antrenare a modelelor tot în Python datorită multitudinii de librării

disponibile.

Personal, m-am confruntat cu lipsa subtitrărilor în filme și tutoriale și consider că această problemă nu este încă rezolvată. Cu cât domeniul de recunoaștere a vorbirii evoluează mai mult și apar seturi de date pentru mai multe limbi, cu atât mai multe persoane vor beneficia de acces la materiale video care nu sunt în limba lor maternă. De asemenea, consider că procesul manual prin care omul adaugă subtitrări este unul consumator de timp și resurse, iar o soluție precum cea prezentată în această lucrare poate simplifica munca depusă.

În concluzie, lucrarea de licență prezintă un prototip de aplicație web care integrează recunoașterea vorbirii și generarea subtitrărilor în contextul videoclip-urilor și oferă utilizatorilor o experiență mai bună în înțelegerea conținutului video.

## 4.2 Perspective

În timpul dezvoltării am observat câteva aspecte care pot fi îmbunătățite pentru o performanță mai bună a aplicației. Printre acestea se numără:

- **Suprimarea zgomotului:** am observat că unele videoclip-uri au melodii sau zgomote de fundal care afectează performanța modelului de recunoaștere a vorbirii. Suprimarea acestor sunete ar putea îmbunătăți calitatea subtitrărilor.
- **Traducerea subtitrărilor:** m-am concentrat pe limba engleză, având mai multe resurse disponibile, dar consider că modelul de recunoaștere a vorbirii poate fi folosit și pentru alte limbi, fie prin antrenarea unui model separat, fie prin traducerea subtitrărilor generate.
- **Seturi de date mai mari:** am folosit seturi de date mici (*MiniLibriSpeech*, *Common Voice Delta Segment 16.1*) din lipsa resurselor computaționale de antrenare, dar consider că antrenarea pe seturi de date mai mari ar îmbunătăți performanța modelului.
- **Topicuri mai diverse:** videoclip-urile sunt clasificate în 5 categorii (*Tech*, *Sport*, *Business*, *Politics*, *Entertainment*), dar natura materialelor video este mult mai diversă de atât. O clasificare mai detaliată ar îmbunătăți experiența utilizatorilor.
- **Spell Checker:** deși am îmbunătățit modelul de recunoaștere a vorbirii cu n-grame, tot mai apar cuvinte greșite. Un spell checker ar putea corecta aceste greșeli.
- **Suport pentru subtitrări:** librăria *transformers* de pe Huggingface nu oferă suport pentru generarea subtitrărilor și a trebuit să prelucrez separat momentele de timp ale cuvintelor și să le sincronizez cu videoclip-ul. În implementarea mea am



ales să grupez câte 7 cuvinte într-o subtitrare, dar în mod normal, această valoare ar trebui să fie ajustată la viteza de vorbire.

- **Procesarea paralelă a secvențelor audio:** după cum am menționat mai sus, fiecare secvență audio este împărțită în bucăți de 30 de secunde, iar aceste bucăți sunt procesate secvențial. Nefiind dependente între ele, pot fi procesate în paralel.
- **Load Balancing:** fiind un prototip, am folosit un singur server pentru toate serviciile, dar într-un mediu de producție, arhitecturi precum *nginx* sau *Kubernetes* sunt vitale pentru existența aplicației.
- **Replici pentru bazele de date:** în aceeași manieră ca la punctul anterior, am folosit o singură bază de date MongoDB și Elasticsearch, dar în producție, arhitecturi de tip *Master-Slave* sau *Sharding* îmbunătățesc viteza de răspuns.
- **Autentificare sigură:** am folosit autentificare cu token JWT, dar în producție, servicii de autentificare precum *Auth0* sau *Firebase* sunt mai sigure.

Și lista poate continua. Procesul de dezvoltare adoptat a fost unul concentrat pe finalizarea funcționalităților promise, dar intrând în detalii, sunt multe aspecte care pot fi îmbunătățite.

# Bibliografie

- [1] Rosana Ardila, Megan Branson, Kelly Davis, Michael Henretty, Michael Kohler, Josh Meyer, Reuben Morais, Lindsay Saunders, Francis M. Tyers și Gregor Weber, *Common Voice: A Massively-Multilingual Speech Corpus*, 2020, arXiv: [1912.06670 \[cs.CL\]](#).
- [2] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed și Michael Auli, *wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations*, 2020, arXiv: [2006.11477 \[cs.CL\]](#).
- [3] Pamela Bump, „How Video Consumption is Changing in 2023”, în *HubSpot* (Apr. 2023), URL: <https://blog.hubspot.com/marketing/how-video-consumption-is-changing>.
- [4] Jacob Devlin, Ming-Wei Chang, Kenton Lee și Kristina Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019, arXiv: [1810.04805 \[cs.CL\]](#).
- [5] Elasticsearch, URL: <https://www.elastic.co/>.
- [6] Jay Gopalakrishnan, „Elasticsearch: What it is, How it works, and what it’s used for”, URL: <https://www.knowi.com/blog/what-is-elastic-search/>.
- [7] Derek Greene și Pádraig Cunningham, „Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering”, *Proc. 23rd International Conference on Machine learning (ICML’06)*, ACM Press, 2006, pp. 377–384.
- [8] Kenneth Heafield, „KenLM: Faster and Smaller Language Model Queries”, *Proceedings of the Sixth Workshop on Statistical Machine Translation*, ed. de Chris Callison-Burch, Philipp Koehn, Christof Monz și Omar F. Zaidan, Edinburgh, Scotland: Association for Computational Linguistics, Iul. 2011, pp. 187–197, URL: <https://aclanthology.org/W11-2123>.
- [9] Eric Jang, Shixiang Gu și Ben Poole, *Categorical Reparameterization with Gumbel-Softmax*, 2017, arXiv: [1611.01144 \[stat.ML\]](#).
- [10] Vassil Panayotov, Guoguo Chen, Daniel Povey și Sanjeev Khudanpur, „Librispeech: An ASR corpus based on public domain audio books” (2015), pp. 5206–5210, DOI: [10.1109/ICASSP.2015.7178964](#).

- [11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai și Soumith Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, 2019, arXiv: [1912.01703 \[cs.LG\]](#).
- [12] Steffen Schneider, Alexei Baevski, Ronan Collobert și Michael Auli, *wav2vec: Unsupervised Pre-training for Speech Recognition*, 2019, arXiv: [1904.05862 \[cs.CL\]](#).
- [13] Jörg Tiedemann, „Parallel Data, Tools and Interfaces in OPUS”, *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, ed. de Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk și Stelios Piperidis, Istanbul, Turkey: European Language Resources Association (ELRA), Mai 2012, pp. 2214–2218, URL: [http://www.lrec-conf.org/proceedings/lrec2012/pdf/463\\_Paper.pdf](http://www.lrec-conf.org/proceedings/lrec2012/pdf/463_Paper.pdf).
- [14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser și Illia Polosukhin, *Attention Is All You Need*, 2023, arXiv: [1706.03762 \[cs.CL\]](#).