Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

\_\_\_\_\_\_\_\_\_\_\_ Едуард ЖАРІКОВ

“\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2025 р.

Веб застосунок для автоматичного підбору вакансій на основі резюме та адаптації резюме за допомогою нейромереж для IT-галузі (комплексна тема)

**Текст програми**

КПІ.ІП-1124.045440.03.12

“ПОГОДЖЕНО”

Керівник проєкту:

\_\_\_\_\_\_\_\_\_\_\_\_ Катерина ЛІЩУК

|  |  |
| --- | --- |
| Нормоконтроль: | Виконавець: |
| \_\_\_\_\_\_\_\_\_\_\_ Катерина ЛІЩУК | \_\_\_\_\_\_\_\_\_\_ Кирил СІДАК |

Київ – 2025

**Посилання на репозиторій з повним текстом програмного коду**

https://github.com/YuraRiabov/KolybaResume

**Файл classification\_trainer.py**

Реалізація функціональної задачі розробкии нейромережі для класифікації резюме

import torch

from torch.utils.data import DataLoader

from torch.nn import Module

from torch.optim import Optimizer

from typing import Callable

from torchmetrics import Metric

from transformers import get\_linear\_schedule\_with\_warmup

from tqdm import tqdm

class ClassificationTrainer:

def \_\_init\_\_(

self,

train\_dataloader: DataLoader,

val\_dataloader: DataLoader,

model: Module,

optimizer: Optimizer,

loss\_function: Callable[[torch.Tensor, torch.Tensor], torch.Tensor],

metric: Metric,

epochs: int,

device: torch.device):

self.train\_dataloader = train\_dataloader

self.val\_dataloader = val\_dataloader

self.model = model

self.optimizer = optimizer

self.loss\_function = loss\_function

self.metric = metric

self.epochs = epochs

self.device = device

self.model.to(self.device)

self.metric.to(self.device)

num\_training\_steps = self.epochs \* len(self.train\_dataloader)

self.scheduler = get\_linear\_schedule\_with\_warmup(self.optimizer, num\_warmup\_steps=0,

num\_training\_steps=num\_training\_steps)

self.history = {'train\_loss': [], 'val\_loss': [], 'train\_metric': [], 'val\_metric': []}

def \_train\_step(self) -> tuple[float, float]:

self.model.train()

loss = 0

for batch in tqdm(self.train\_dataloader, desc="Training", leave=False):

token\_ids = batch['input\_ids'].to(self.device)

attention\_mask = batch['attention\_mask'].to(self.device)

labels = batch['labels'].to(self.device)

self.optimizer.zero\_grad()

batch\_loss, logits = self.model(input\_ids=token\_ids, attention\_mask=attention\_mask, labels=labels,

return\_dict=False)

loss += batch\_loss.item()

batch\_loss.backward()

torch.nn.utils.clip\_grad\_norm\_(self.model.parameters(), 1.0)

self.optimizer.step()

self.scheduler.step()

preds = torch.argmax(logits, dim=1)

self.metric.update(preds, labels)

avg\_loss = loss / len(self.train\_dataloader)

metric\_result = self.metric.compute()

return avg\_loss, metric\_result.item()

def \_val\_step(self) -> tuple[float, float]:

self.model.eval()

loss = 0

with torch.no\_grad():

for batch in tqdm(self.val\_dataloader, desc="Validating", leave=False):

token\_ids = batch['input\_ids'].to(self.device)

attention\_mask = batch['attention\_mask'].to(self.device)

labels = batch['labels'].to(self.device)

batch\_loss, logits = self.model(input\_ids=token\_ids, attention\_mask=attention\_mask, labels=labels,

return\_dict=False)

loss += batch\_loss.item()

preds = torch.argmax(logits, dim=1)

self.metric.update(preds, labels)

avg\_loss = loss / len(self.val\_dataloader)

metric\_result = self.metric.compute()

return avg\_loss, metric\_result.item()

def fine\_tune(self) -> None:

for epoch in range(1, self.epochs + 1):

self.metric.reset()

train\_loss, train\_metric = self.\_train\_step()

self.metric.reset()

val\_loss, val\_metric = self.\_val\_step()

self.history['train\_loss'].append(train\_loss)

self.history['val\_loss'].append(val\_loss)

self.history['train\_metric'].append(train\_metric)

self.history['val\_metric'].append(val\_metric)

print(f'Epoch {epoch}/{self.epochs}')

print(f' Train Loss: {train\_loss:.4f}, Train Metric: {train\_metric:.4f}')

print(f' Val Loss: {val\_loss:.4f}, Val Metric: {val\_metric:.4f}')

**Файл train\_bert\_classifier.py**

Реалізація функціональної задачі розробкии нейромережі для класифікації резюме

from torch.optim import AdamW

from torch.utils.data import DataLoader

from transformers import BertTokenizer, BertForSequenceClassification

from ml\_backend.resume\_classifier import ClassificationTrainer, ResumeDataset

from sklearn.preprocessing import LabelEncoder

from torchmetrics import F1Score

from sklearn.model\_selection import train\_test\_split

import pandas as pd

import torch

from joblib import dump

file\_path = "/content/drive/MyDrive/preprocessed\_resumes.parquet"

EPOCHS = 2

BATCH\_SIZE = 16

LEARNING\_RATE = 2e-5

MODEL\_NAME = 'bert-base-uncased'

MAX\_LENGTH = 512

VAL\_SIZE = 0.1

RANDOM\_STATE = 42

def load\_data(file\_path: str) -> tuple[list[str], list[int], LabelEncoder]:

df = pd.read\_parquet(file\_path)

resumes = df['Resume'].tolist()

encoder = LabelEncoder()

labels = encoder.fit\_transform(df['Category']).tolist()

return resumes, labels, encoder

resumes, labels, encoder = load\_data(file\_path)

train\_resumes, val\_resumes, train\_labels, val\_labels = train\_test\_split(

resumes, labels, test\_size=VAL\_SIZE, stratify=labels, random\_state=42)

tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

train\_dataset = ResumeDataset(train\_resumes, train\_labels, tokenizer, max\_length=MAX\_LENGTH)

val\_dataset = ResumeDataset(val\_resumes, val\_labels, tokenizer, max\_length=MAX\_LENGTH)

train\_dataloader = DataLoader(train\_dataset, batch\_size=BATCH\_SIZE, shuffle=True)

val\_dataloader = DataLoader(val\_dataset, batch\_size=BATCH\_SIZE, shuffle=False)

num\_classes = len(encoder.classes\_)

model = BertForSequenceClassification.from\_pretrained(

'bert-base-uncased',

num\_labels=num\_classes)

optimizer = AdamW(model.parameters(), lr=LEARNING\_RATE)

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

loss\_function = torch.nn.CrossEntropyLoss()

metric = F1Score(task='multiclass', num\_classes=num\_classes, average='macro')

trainer = ClassificationTrainer(

train\_dataloader=train\_dataloader,

val\_dataloader=val\_dataloader,

model=model,

optimizer=optimizer,

loss\_function=loss\_function,

metric=metric,

epochs=EPOCHS,

device=device

)

trainer.fine\_tune()

model.save\_pretrained("/content/drive/MyDrive/fine\_tuned\_bert")

tokenizer.save\_pretrained("/content/drive/MyDrive/fine\_tuned\_bert")

dump(encoder, "/content/drive/MyDrive/label\_encoder.joblib")

**Файл vacancy\_service.py**

Реалізація функціональної задачі розробки методів автоматизованого пошуку релевантних вакансій

from sqlalchemy import select

from sqlalchemy.orm import Session

from ml\_backend.api.db.models import Vacancy, Resume

from ml\_backend.api.models.schemas import VacancyScoreResponse, ResumeVacancyMatch

from ml\_backend.api.services.cleaning\_service import clean\_text, translate

from ml\_backend.api.services.model\_service import get\_embedding\_model

from sklearn.metrics.pairwise import cosine\_similarity

import numpy as np

from collections import defaultdict

import logging

logger = logging.getLogger(\_\_name\_\_)

def process\_vacancies(db: Session, vacancy\_ids: list[int], batch\_size: int = 32) -> list[VacancyScoreResponse]:

stmt = select(Vacancy).where(Vacancy.Id.in\_(vacancy\_ids))

all\_vacancies = db.execute(stmt).scalars().all()

if not all\_vacancies:

return []

model = get\_embedding\_model()

vacancy\_vectors = {}

for i in range(0, len(all\_vacancies), batch\_size):

batch\_vacancies = all\_vacancies[i:i + batch\_size]

batch\_texts = []

successful\_vacancies = []

for vacancy in batch\_vacancies:

try:

text = translate(f"{vacancy.Title} {vacancy.Text}")

cleaned = clean\_text(text)

vacancy.CleanedText = cleaned

batch\_texts.append(cleaned)

successful\_vacancies.append(vacancy)

except Exception as e:

logger.error(f'Error processing vacancy {vacancy.Id}: {str(e)}')

if not successful\_vacancies:

continue

try:

encoded\_batch = model.encode(batch\_texts)

for j, vacancy in enumerate(successful\_vacancies):

vector = encoded\_batch[j]

vacancy.Vector = vector.tobytes()

vacancy\_vectors[vacancy.Id] = vector

db.commit()

logger.info(f'Processed {len(successful\_vacancies)} vacancies')

except Exception as e:

db.rollback()

logger.error(f'Error encoding batch: {str(e)}')

categories = {v.Category for v in all\_vacancies if v.Category is not None}

if not categories:

return []

stmt = select(Resume).where(

Resume.Category.in\_(categories),

Resume.Vector.is\_not(None)

)

matching\_resumes = db.execute(stmt).scalars().all()

if not matching\_resumes:

return []

resumes\_by\_category = defaultdict(list)

resume\_vectors = {}

for resume in matching\_resumes:

resumes\_by\_category[resume.Category].append(resume)

resume\_vectors[resume.Id] = np.frombuffer(resume.Vector, dtype=np.float32)

results = []

for vacancy in all\_vacancies:

vacancy\_category = vacancy.Category

if vacancy\_category not in resumes\_by\_category:

continue

vacancy\_vector = vacancy\_vectors[vacancy.Id]

for resume in resumes\_by\_category[vacancy\_category]:

resume\_vector = resume\_vectors[resume.Id]

similarity = model.similarity(vacancy\_vector, resume\_vector)[0][0]

score = int(similarity \* 100)

results.append(VacancyScoreResponse(

user\_id=resume.UserId,

vacancy\_id=vacancy.Id,

score=score

))

return results

def get\_matches\_for\_resume(db: Session, resume: Resume) -> list[ResumeVacancyMatch]:

if not resume.Vector or not resume.Category:

return []

stmt = select(Vacancy).where(

Vacancy.Category == resume.Category,

Vacancy.Vector.is\_not(None)

)

result = db.execute(stmt)

matching\_vacancies = result.scalars().all()

if not matching\_vacancies:

return []

resume\_vector = np.frombuffer(resume.Vector, dtype=np.float32)

results = []

for vacancy in matching\_vacancies:

vacancy\_vector = np.frombuffer(vacancy.Vector, dtype=np.float32)

similarity = cosine\_similarity(

resume\_vector.reshape(1, -1),

vacancy\_vector.reshape(1, -1)

)[0][0]

score = int(similarity \* 100)

results.append(

ResumeVacancyMatch(

vacancy\_id=vacancy.Id,

score=score

)

)

results.sort(key=lambda x: x.score, reverse=True)

return results[:200]

**Файл adaptation\_service.py**

Реалізація функціональної задачі розробки алгоритму для адаптації резюме import json

from ml\_backend.api.services.keyword\_service import extract\_keywords

from ml\_backend.api.services.model\_service import get\_embedding\_model

from ml\_backend.api.db.models import Resume

from ml\_backend.api.models.schemas import AdaptationResponse

from ml\_backend.api.services.cleaning\_service import clean\_text, translate

def get\_keywords\_score(resume: Resume, vacancy\_text: str, clean: bool) -> AdaptationResponse:

if clean:

vacancy\_text = translate(vacancy\_text)

vacancy\_text = clean\_text(vacancy\_text)

resume\_keywords = set(json.loads(resume.Keywords))

vacancy\_keywords, vacancy\_skills = extract\_keywords(vacancy\_text, extract\_skills=True)

missing\_keywords = list(vacancy\_skills - resume\_keywords)

score = keyword\_similarity(resume\_keywords, vacancy\_keywords)

return AdaptationResponse(score=score, missing\_keywords=missing\_keywords)

def keyword\_similarity(resume\_keywords: set[str], vacancy\_keywords: set[str]) -> int:

resume\_key\_text = " ".join(resume\_keywords)

vacancy\_key\_text = " ".join(vacancy\_keywords)

model = get\_embedding\_model()

resume\_embedding = model.encode(resume\_key\_text)

vacancy\_embedding = model.encode(vacancy\_key\_text)

similarity = model.similarity(resume\_embedding, vacancy\_embedding)[0][0]

return int(similarity \* 100)

**Файл keyword\_service.py**

Реалізація функціональної задачі розробки алгоритму для адаптації резюме

from transformers import Pipeline, BertTokenizer

from ml\_backend.api.services.model\_service import get\_keybert\_model, get\_skills\_model

def create\_overlapping\_chunks(text: str, tokenizer: BertTokenizer, max\_length: int = 512, overlap: int = 50) -> list[

str]:

full\_tokens = tokenizer.encode(text, add\_special\_tokens=False, truncation=False)

total\_tokens = len(full\_tokens)

effective\_max\_length = max\_length - 2

if total\_tokens <= effective\_max\_length:

return [text]

chunks = []

start\_token = 0

while start\_token < total\_tokens:

end\_token = min(start\_token + effective\_max\_length, total\_tokens)

chunk\_tokens = full\_tokens[start\_token:end\_token]

chunk\_text = tokenizer.decode(chunk\_tokens, skip\_special\_tokens=True)

chunks.append(chunk\_text)

if end\_token >= total\_tokens:

break

start\_token = end\_token - overlap

return chunks

def extract\_chunk\_skills(chunk\_text: str, pipe: Pipeline) -> set[str]:

results = pipe(chunk\_text)

skills = []

current\_skill\_words = []

prev\_end = 0

for result in results:

entity = result['entity']

word = result['word']

start = result['start']

end = result['end']

if entity.startswith('B-SKILL') and result['score'] > 0.6:

if current\_skill\_words:

skill = ''.join(current\_skill\_words).replace('##', '')

skill = skill.strip()

if skill and len(skill) > 1:

skills.append(skill)

current\_skill\_words = [word]

elif entity.startswith('I-SKILL') and current\_skill\_words:

word = word if start == prev\_end else ' ' + word

current\_skill\_words.append(word)

else:

if current\_skill\_words:

skill = ''.join(current\_skill\_words).replace('##', '').strip()

if skill and len(skill) > 1:

skills.append(skill)

current\_skill\_words = []

prev\_end = end

if current\_skill\_words:

skill = ''.join(current\_skill\_words).replace('##', '').strip()

if skill:

skills.append(skill)

skills = set([skill.lower() for skill in skills])

return skills

def is\_repeated\_word(phrase: str) -> bool:

words = phrase.lower().split()

return len(words) > 1 and all(word == words[0] for word in words)

def extract\_keywords(text: str, extract\_skills: bool = False, top\_n: int = 100, max\_tokens: int = 510,

overlap\_tokens: int = 50) -> set[str] | tuple[set[str], set[str]]:

tokenizer, pipe = get\_skills\_model()

chunks = create\_overlapping\_chunks(text, tokenizer, max\_tokens, overlap\_tokens)

model = get\_keybert\_model()

unigrams = model.extract\_keywords(

chunks,

keyphrase\_ngram\_range=(1, 1),

use\_mmr=True,

diversity=0.7,

top\_n=top\_n

)

bigrams = model.extract\_keywords(

chunks,

keyphrase\_ngram\_range=(2, 2),

use\_mmr=True,

diversity=0.7,

top\_n=top\_n

)

all\_keywords = unigrams + bigrams

if all\_keywords and isinstance(all\_keywords[0], tuple):

all\_keywords = [all\_keywords]

filtered\_keywords = set()

for keywords in all\_keywords:

for keyword, \_ in keywords:

if not is\_repeated\_word(keyword):

filtered\_keywords.add(keyword.lower())

if extract\_skills:

skills = set()

for chunk in chunks:

chunk\_skills = extract\_chunk\_skills(chunk, pipe)

skills = skills.union(chunk\_skills)

skills = filtered\_keywords.intersection(skills)

return filtered\_keywords, skills

return filtered\_keywords

**Файл resume.py**

Реалізація функціональної задачі проектування та розробки API

from fastapi import APIRouter, Depends, HTTPException, status

from sqlalchemy.orm import Session

from ml\_backend.api.models.schemas import ResumeRequest

from ml\_backend.api.db.base import get\_db

from ml\_backend.api.services.resume\_service import preprocess\_resume\_text

from ml\_backend.api.db.models import Resume

router = APIRouter()

@router.put("/resume", status\_code=status.HTTP\_200\_OK)

async def process\_resume(request: ResumeRequest, db: Session = Depends(get\_db)):

resume = db.get(Resume, request.resume\_id)

if not resume:

raise HTTPException(

status\_code=status.HTTP\_404\_NOT\_FOUND,

detail=f"Resume with id {request.resume\_id} not found"

)

preprocess\_resume\_text(db, resume)

return {"status": "success", "message": "Resume vector updated successfully"}

**Файл vacancies.py**

Реалізація функціональної задачі проектування та розробки API

from fastapi import APIRouter, Depends, HTTPException, status

from sqlalchemy.orm import Session

from ml\_backend.api.db.models import Resume

from ml\_backend.api.models.schemas import VacanciesRequest, VacancyScoreResponse, ResumeVacancyMatch

from ml\_backend.api.db.base import get\_db

from ml\_backend.api.services.vacancy\_service import process\_vacancies, get\_matches\_for\_resume

import logging

logger = logging.getLogger(\_\_name\_\_)

router = APIRouter()

@router.post("/vacancies", response\_model=list[VacancyScoreResponse])

async def process\_new\_vacancies(request: VacanciesRequest, db: Session = Depends(get\_db)):

if not request.vacancy\_ids:

raise HTTPException(

status\_code=status.HTTP\_400\_BAD\_REQUEST,

detail="No vacancy IDs provided"

)

results = process\_vacancies(db, request.vacancy\_ids)

return results

@router.get("/vacancies/score/{resume\_id}", response\_model=list[ResumeVacancyMatch])

async def get\_vacancy\_matches(resume\_id: int, db: Session = Depends(get\_db)):

resume = db.get(Resume, resume\_id)

if not resume:

raise HTTPException(status.HTTP\_404\_NOT\_FOUND, detail=f"Resume with {resume\_id} not found")

matches = get\_matches\_for\_resume(db, resume)

return matches

**Файл adaptation.py**

Реалізація функціональної задачі проектування та розробки API

from fastapi import APIRouter, Depends, HTTPException, status

from sqlalchemy.orm import Session

from ml\_backend.api.models.schemas import AdaptationRequest, AdaptationResponse

from ml\_backend.api.db.base import get\_db

from ml\_backend.api.services.adaptation\_service import get\_keywords\_score

from ml\_backend.api.db.models import Resume

router = APIRouter()

@router.post("/adaptation", response\_model=AdaptationResponse)

async def get\_recommendations(request: AdaptationRequest, db: Session = Depends(get\_db)):

if not request.vacancy\_text.strip():

raise HTTPException(

status\_code=status.HTTP\_400\_BAD\_REQUEST,

detail=f"Vacancy text cannot be empty"

)

resume = db.get(Resume, request.resume\_id)

if not resume:

raise HTTPException(

status\_code=status.HTTP\_404\_NOT\_FOUND,

detail=f"Resume with id {request.resume\_id} not found"

)

result = get\_keywords\_score(resume, request.vacancy\_text, request.clean)

return result