Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

“ЗАТВЕРДЖЕНО”

Завідувач кафедри

_____ Едуард ЖАРІКОВ

“___” _____ 2025 р.

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

API машинного навчання

**Текст програми**

КПІ.ІП-1124.045440.03.12

“ПОГОДЖЕНО”

Керівник проєкту:

_____ Катерина ЛІЩУК

Нормоконтроль:

_____ Катерина ЛІЩУК

Виконавець:

_____ Кирил СІДАК

Київ – 2025

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

https://github.com/YuraRiabov/KolybaResume

**Файл classification_trainer.py**

Реалізація функціональної задачі розробки 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 оцінки якості навченої моделі 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 методу 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 методу 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 методу 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 машинного навчання

```
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 машинного навчання

```
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 машинного навчання

```
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
```