ДОДАТОК А

Програмний код для парсингу веб-сторінок

та аналізу отриманого контенту

```
import os
import time
import mimetypes
import re
from dotenv import load_dotenv
from celery import Celery
from minio.error import S3Error

from aws_s3 import aws_s3_full_host, upload_to_s3,
download_from_s3
from helpers import send_to_webhook
from keywords import get_keywords,
get_keywords_for_sentences
from music import get_music
from ner import parse_ner
from pexels import retrieve_videos_for_scenes,
download_video
from playwright_helpers import scrape_with_playwright_sync
from summarizer import get_summary
from text_analysis import get_sentiment, get_category
from translator import Translator
from video_generator import VideoGenerator, Scene,
get_video_generator
from voiceover import generate_speech_from_sentences

celery = Celery('ai-worker',
broker='redis://localhost:6379/0')
celery.conf.update(result_backend='redis://localhost:6379/
0')

load_dotenv()


@celery.task
def parse_url_task(web_page_uuid, url, webhook_url):
    content, article, file_name, lang =
scrape_with_playwright_sync(url)
    new_file_name = f"{web_page_uuid}.jpeg"
    os.rename(file_name, new_file_name)
```

```
        destination = f"web-page-screenshots/{new_file_name}"

        mime_type, _ = mimetypes.guess_type(new_file_name)

        if mime_type is None:
            mime_type = 'application/octet-stream'

        try:
            upload_to_s3(new_file_name, destination,
mime_type)
            screenshot_url = aws_s3_full_host + '/' +
destination

            os.remove(new_file_name)

            send_to_webhook(webhook_url, {
                "event": "webpage_screenshot",
                "payload": {
                    "entity": "web_page",
                    "uuid": web_page_uuid
                },
                "data": {
                    "screenshot_url": screenshot_url
                }
            })
        except S3Error as exc:
            print("Error occurred.", exc)
            return "Error occurred while uploading"

        time.sleep(3)

        article.nlp()

        authors = article.authors
        if authors:
            authors = ', '.join(authors)
        else:
            authors = None

        published_at = article.publish_date
        if published_at:
            published_at = published_at.isoformat() + "Z"
        else:
            published_at = None

        summary = get_summary(article.text, lang)
```

```
        if summary is None:
            summary = article.summary

        if lang == "uk":
            translator = Translator(use_google=True)
            translated_text = translator.translate(summary,
"uk", "en")
            sentiment = get_sentiment(translated_text)
            keywords = get_keywords(summary)
            keywords = [translator.translate(keyword, "en",
"uk") for keyword in keywords]
            category = get_category(translated_text)
        else:
            sentiment = get_sentiment(summary)
            keywords = get_keywords(summary)
            category = get_category(summary)

        parsed_summary = parse_ner(text=summary, lang=lang)

        send_to_webhook(webhook_url, {
            "event": "webpage_data",
            "payload": {
                "entity": "web_page",
                "uuid": web_page_uuid
            },
            "data": {
                "title": article.title,
                "url": url,
                "language": lang,
                "site_name": article.source_url,
                "author": authors,
                "images": article.images,
                "content": article.text,
                "summary": parsed_summary,
                "sentiment": sentiment,
                "keywords": keywords,
                "categories": [category],
                "published_at": published_at
            }
        })

        return url

    from playwright.async_api import async_playwright
    from playwright.sync_api import sync_playwright
```

```
    import newspaper
    import time
    from urllib.parse import urlparse
    import markdownify
    from langdetect import detect


    async def validate_url(url):
        playwright = await async_playwright().start()
        browser = await
playwright.chromium.launch(headless=True)
        page = await browser.new_page()

        await page.goto(url)

        time.sleep(1)

        content = await page.content()
        await browser.close()

        markdown_content =
markdownify.markdownify(str(content))
        lang = detect(markdown_content)

        return lang in ['en', 'uk']


    def scrape_with_playwright_sync(url):
        with sync_playwright() as playwright:
            browser =
playwright.chromium.launch(headless=True)
            page = browser.new_page()

            page.goto(url)

            page.wait_for_load_state('domcontentloaded')

            domain = urlparse(url).netloc.replace('.', '-')
            file_name = f"{time.time()}-{domain}.jpeg"

            page.screenshot(path=file_name)

            content = page.content()

            browser.close()
```

```
            markdown_content =
markdownify.markdownify(content)

            lang = detect(markdown_content)

            article = newspaper.Article(url,
input_html=content, language=lang, fetch_images=True)
            article.download()
            article.parse()

            return content, article, file_name, lang

    import nltk
    from nltk.corpus import stopwords
    from nltk.cluster.util import cosine_distance
    from nltk.tokenize import sent_tokenize
    import numpy as np
    import networkx as nx
    import re
    import os
    from dotenv import load_dotenv
    from transformers import pipeline, BartTokenizer,
BartForConditionalGeneration
    import requests
    import json

    nltk.download('stopwords')
    nltk.download('punkt')

    load_dotenv()


    def read_article(text):
        sentences = []
        sentences = sent_tokenize(text)
        for sentence in sentences:
            sentence.replace("[^a-zA-Z0-9]", " ")
        return sentences


    def sentence_similarity(sent1, sent2, stopwords=None):
        if stopwords is None:
            stopwords = []
        sent1 = [w.lower() for w in sent1]
        sent2 = [w.lower() for w in sent2]
```

```
        all_words = list(set(sent1 + sent2))

        vector1 = [0] * len(all_words)
        vector2 = [0] * len(all_words)
        # build the vector for the first sentence
        for w in sent1:
            if not w in stopwords:
                vector1[all_words.index(w)] += 1
                # build the vector for the second sentence
        for w in sent2:
            if not w in stopwords:
                vector2[all_words.index(w)] += 1

        return 1 - cosine_distance(vector1, vector2)


    def build_similarity_matrix(sentences, stop_words):
        similarity_matrix = np.zeros((len(sentences),
len(sentences)))
        for idx1 in range(len(sentences)):
            for idx2 in range(len(sentences)):
                if idx1 != idx2:
                    similarity_matrix[idx1][idx2] =
sentence_similarity(sentences[idx1], sentences[idx2],
stop_words)
        return similarity_matrix


    def get_extractive_summary(language, text, top_n):
        summarize_text = []
        sentences = read_article(text)
        sentence_similarity_matrix =
build_similarity_matrix(sentences, None)
        sentence_similarity_graph =
nx.from_numpy_array(sentence_similarity_matrix)
        scores = nx.pagerank(sentence_similarity_graph)
        ranked_sentences = sorted(((scores[i], s) for i, s in
enumerate(sentences)), reverse=True)

        for i in range(top_n):
            summarize_text.append(ranked_sentences[i][1])

        return " ".join(summarize_text), len(sentences)


    def get_abstractive_summary(text):
```

```
        bart_tokenizer =
BartTokenizer.from_pretrained('facebook/bart-large-cnn')
        bart_model =
BartForConditionalGeneration.from_pretrained('facebook/bart-
large-cnn')
        """Function to get summary by bart model"""
        input_ids = bart_tokenizer.encode(text,
return_tensors="pt", max_length=512)
        summary_ids = bart_model.generate(input_ids,
max_length=200, min_length=12,
                                          length_penalty=1.0,
num_beams=4,
                                          early_stopping=True)
        output_summ = [bart_tokenizer.decode(g,
skip_special_tokens=True,

clean_up_tokenization_spaces=False)
                       for g in summary_ids]
        return output_summ[0]



    from transformers import AutoTokenizer,
AutoModelForTokenClassification
    from transformers import pipeline


    def parse_ner(text, lang='en'):
        if lang == 'uk':
            tokenizer =
AutoTokenizer.from_pretrained("EvanD/xlm-roberta-base-
ukrainian-ner-ukrner")
            ner_model =
AutoModelForTokenClassification.from_pretrained("EvanD/xlm-
roberta-base-ukrainian-ner-ukrner")
            ner = pipeline("ner", model=ner_model,
tokenizer=tokenizer, aggregation_strategy="simple")
        else:
            tokenizer =
AutoTokenizer.from_pretrained("Davlan/xlm-roberta-base-ner-
hrl")
            model =
AutoModelForTokenClassification.from_pretrained("Davlan/xlm-
roberta-base-ner-hrl")
            ner = pipeline("ner", model=model,
tokenizer=tokenizer)
```

```
        ner_result = ner(text)

        return replace_named_entities(text, ner_result,
lang=lang)


    def replace_named_entities(text, entities, lang='en'):
        # If English model, we need to merge 'B-' and 'I-'
tokens
        merged_entities = []
        if lang == 'en':
            current_entity = None
            for ent in entities:
                label = ent['entity']
                word = ent['word'].replace('▁', ' ')
                if label.startswith('B-'):
                    if current_entity:
                        merged_entities.append(current_entity)
                    current_entity = {
                        'entity_group': label[2:],
                        'start': ent['start'] + 1,
                        'end': ent['end'],
                        'word': word.strip()
                    }
                elif label.startswith('I-') and
current_entity:
                    current_entity['end'] = ent['end']
                    current_entity['word'] += word
                else:
                    if current_entity:
                        merged_entities.append(current_entity)
                        current_entity = None
            if current_entity:
                merged_entities.append(current_entity)
        else:
            merged_entities = entities

        merged_entities = sorted(merged_entities, key=lambda
x: x['start'])

        result = []
        last_idx = 0
        for ent in merged_entities:
            start = ent['start']
            end = ent['end']
```

```
        entity_text = text[start:end]

        if last_idx < start:
            result.append(text[last_idx:start])

        result.append(f"@{{{entity_text}}}")

        last_idx = end

    if last_idx < len(text):
        result.append(text[last_idx:])

    return ''.join(result)
```

ДОДАТОК Б

Програмний код для нейронної мережі

для виявлення категорії тексту та її навчання

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import tensorflow as tf

data = pd.read_csv('processed_data.csv')
X = data.iloc[:,0]
y = data.iloc[:,1]

X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=0, test_size=0.3)
X_test, X_val, y_test, y_val = train_test_split(X_test,
y_test, random_state=0, test_size=0.7)
max_length = np.max(X_train.apply(lambda x: len(x)))

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import
pad_sequences

tokenizer = Tokenizer()
tokenizer_predict = Tokenizer()
tokenizer.fit_on_texts(X_train)

vocab_length = len(tokenizer.word_index) + 1

X_train = tokenizer.texts_to_sequences(X_train)
X_val = tokenizer.texts_to_sequences(X_val)
X_test = tokenizer.texts_to_sequences(X_test)

X_train = pad_sequences(X_train, maxlen=max_length,
padding='post')
X_val = pad_sequences(X_val, maxlen=max_length,
padding='post')
X_test = pad_sequences(X_test, maxlen=max_length,
padding='post')

y_train.values.reshape(-1, 1), y_train.shape
y_val.values.reshape(-1, 1), y_val.shape
```

```
    y_test.values.reshape(-1, 1), y_test.shape

    max_category_number = max(y_train)

    from tensorflow.keras.utils import to_categorical

    category_count = max_category_number + 1
    y_train = to_categorical(y_train, category_count)
    y_val = to_categorical(y_val, category_count)
    y_test = to_categorical(y_test, category_count)

    embedding_dim = 16

    def create_model(gru_units=256, dense_units=256,
dropout_rate=0.4):
        model = tf.keras.Sequential([
            tf.keras.Input(shape=(max_length,)),
            tf.keras.layers.Embedding(vocab_length,
embedding_dim),

tf.keras.layers.Bidirectional(tf.keras.layers.GRU(gru_units,
return_sequences=True)),
            tf.keras.layers.GlobalAveragePooling1D(),
            tf.keras.layers.Dense(dense_units,
activation=tf.keras.activations.tanh),
            tf.keras.layers.Dense(dense_units,
activation=tf.keras.activations.tanh),
            tf.keras.layers.Dense(dense_units,
activation=tf.keras.activations.tanh),
            tf.keras.layers.Dropout(dropout_rate),
            tf.keras.layers.Dense(category_count,
activation='softmax')
        ])
        optimizer =
tf.keras.optimizers.Adam(learning_rate=0.001)
        model.compile(loss='categorical_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
        return model

    from sklearn.model_selection import GridSearchCV
    from scikeras.wrappers import KerasClassifier

    model = KerasClassifier(build_fn=create_model, epochs=5,
batch_size=300)

    param_grid = {
```

```
        'model__gru_units': [128, 256],
        'model__dense_units': [128, 256],
        'model__dropout_rate': [0.2, 0.4, 0.6]
    }

    grid = GridSearchCV(estimator=model,
param_grid=param_grid, cv=3, verbose=2)
    grid_result = grid.fit(X_train, y_train)

    # Summarize results
    print("Best: %f using %s" % (grid_result.best_score_,
grid_result.best_params_))
    means = grid_result.cv_results_['mean_test_score']
    stds = grid_result.cv_results_['std_test_score']
    params = grid_result.cv_results_['params']
    for mean, stdev, param in zip(means, stds, params):
        print("%f (%f) with: %r" % (mean, stdev, param))

    model = create_model(gru_units=256, dense_units=256,
dropout_rate=0.4)

    num_epochs = 12
    early_stopping =
tf.keras.callbacks.EarlyStopping(patience=3,
monitor='val_loss', restore_best_weights=True)
    history = model.fit(X_train, y_train, epochs=num_epochs,
batch_size=256, validation_data=(X_val, y_val),
callbacks=[early_stopping])

    test_loss, test_accuracy = model.evaluate(X_test, y_test)

    print("Test Loss:", test_loss)
    print("Test Accuracy:", test_accuracy)

    model.save("model.keras")
    model.save_weights("model-weights.h5")

    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']

    loss = history.history['loss']
    val_loss = history.history['val_loss']

    epochs = range(len(acc))

    plt.plot(epochs, acc,'b',label='training acc')
```

```
    plt.plot(epochs, val_acc, 'r', label='validation acc')
    plt.legend()
    plt.show()

    plt.plot(epochs, loss,'b',label='training loss')
    plt.plot(epochs, val_loss, 'r', label='validation loss')
    plt.legend()
    plt.show()

    from sklearn.metrics import accuracy_score,
precision_score, recall_score, f1_score

    y_pred = model.predict(X_test)
    y_pred = (y_pred > 0.5)

    accuracy = accuracy_score(y_test, y_pred)
    precision_micro = precision_score(y_test, y_pred,
average='micro')
    recall_micro = recall_score(y_test, y_pred,
average='micro')
    f1_result = f1_score(y_test, y_pred, average='micro')

    print("Multi-Class Classification Metrics:")
    print("- Accuracy:", accuracy)
    print("- Precision:", precision_micro)
    print("- Recall:", recall_micro)
    print("- F1 Score:", f1_result)
```

ДОДАТОК В

Програмний код для створення відео

```
from moviepy.editor import (
    VideoFileClip,
    AudioFileClip,
    CompositeAudioClip,
    TextClip,
    ImageClip,
    ColorClip,
    CompositeVideoClip,
    concatenate_videoclips,
    afx
)
from moviepy.video.fx.all import loop
from PIL import Image, ImageColor


class Scene:
    def __init__(
            self,
            video_path,
            voiceover_path,
            text,
            image_path=None,
            image_caption=None
    ):
        self.video_path = video_path
        self.voiceover_path = voiceover_path
        self.text = text
        self.image_path = image_path
        self.image_caption = image_caption


class VideoGenerator:
    def __init__(self, background_music_path,
background_volume=0.2,
                 text_font='Arial-Bold',
text_color='white', text_size=50,
                 text_bg_color='black', text_align='left',
                 text_margin=(10, 10), text_padding=(10,
10),
                 scene_image_text_bg_color='black',
scene_image_text_color='white',
                 video_width=1920, video_height=1080,
```

```
                    logo_path=None, logo_position=(10, 10),
logo_height=100,
                    portrait_height=100,
portrait_caption_font_size=None,
                    bitrate='5000k', fps=24):
            self.scenes = []
            self.background_music_path = background_music_path
            self.background_volume = background_volume
            self.text_font = text_font
            self.text_color = text_color
            self.text_size = text_size
            self.text_bg_color = text_bg_color
            self.text_align = text_align
            self.text_margin = text_margin  # (margin_x,
margin_y)
            self.text_padding = text_padding  # (padding_x,
padding_y)
            self.scene_image_text_bg_color =
scene_image_text_bg_color
            self.scene_image_text_color =
scene_image_text_color
            self.video_width = video_width
            self.video_height = video_height
            self.logo_path = logo_path
            self.logo_position = logo_position
            self.logo_height = logo_height
            self.portrait_height = portrait_height
            self.portrait_caption_font_size = (
                portrait_caption_font_size
                if portrait_caption_font_size
                else int(self.text_size * 0.6)
            )
            self.bitrate = bitrate
            self.fps = fps

        def add_scene(self, scene):
            self.scenes.append(scene)

        def _create_scene_clip(self, scene):
            voiceover_audio =
AudioFileClip(scene.voiceover_path)
            voiceover_duration = voiceover_audio.duration
            scene_duration = voiceover_duration + 1.0
            raw_video = VideoFileClip(scene.video_path)

            if raw_video.duration < scene_duration:
```

```
                video_clip = raw_video.fx(loop,
duration=scene_duration).without_audio()
            else:
                video_clip = raw_video.subclip(0,
scene_duration).without_audio()
            video_clip =
video_clip.set_duration(scene_duration)

            target_aspect = self.video_width /
self.video_height

            if self.video_height > self.video_width:
                video_clip =
video_clip.resize(height=self.video_height)

                video_clip = video_clip.crop(
                    x_center=video_clip.w / 2,
                    y_center=video_clip.h / 2,
                    width=self.video_width,
                    height=self.video_height
                )
            else:
                source_aspect = video_clip.w / video_clip.h

                if source_aspect > target_aspect:
                    video_clip =
video_clip.resize(height=self.video_height)
                else:
                    video_clip =
video_clip.resize(width=self.video_width)

                video_clip = video_clip.crop(
                    x_center=video_clip.w / 2,
                    y_center=video_clip.h / 2,
                    width=self.video_width,
                    height=self.video_height
                )

            padding_x, padding_y = self.text_padding

            margin_x, margin_y = self.text_margin
            max_text_width = video_clip.w - 2 * margin_x - 2 *
padding_x

            text_pos = {
                'center': 'center',
```

```
                'right': 'East'
            }.get(self.text_align, 'West')

            text_clip = TextClip(
                scene.text,
                fontsize=self.text_size,
                font=self.text_font,
                color=self.text_color,
                bg_color=self.text_bg_color,
                method='caption',
                size=(max_text_width, None),
                align=text_pos
            ).set_duration(scene_duration).to_RGB()

            padded_width = text_clip.w + 2 * padding_x
            padded_height = text_clip.h + 2 * padding_y

            bg_color = ImageColor.getrgb(self.text_bg_color)

            bg_clip = ColorClip(
                size=(padded_width, padded_height),
                color=bg_color
            ).set_duration(scene_duration).to_RGB()

            text_with_padding = CompositeVideoClip(
                [bg_clip, text_clip.set_position(('center',
'center'))],
                size=bg_clip.size
            ).set_duration(scene_duration).to_RGB()

            final_y = video_clip.h - text_with_padding.h -
margin_y

            def sliding_position(t):
                slide_duration = 1.0
                if t < slide_duration:
                    x = -text_with_padding.w + (margin_x +
text_with_padding.w) * (t / slide_duration)
                else:
                    x = margin_x
                return (x, final_y)

            text_with_padding =
text_with_padding.set_position(sliding_position)

            layers = [video_clip, text_with_padding]
```

```
            if scene.image_path:
                image_clip = ImageClip(scene.image_path,
transparent=True).set_duration(scene_duration)
                image_clip =
image_clip.resize(height=self.portrait_height).to_RGB()

                bg_image = ColorClip(
                    size=(image_clip.w, image_clip.h),

color=ImageColor.getrgb(self.scene_image_text_bg_color)
                ).set_duration(scene_duration).to_RGB()
                image_bg_group = CompositeVideoClip(
                    [bg_image,
image_clip.set_position(('center', 'center'))],
                    size=bg_image.size
                ).set_duration(scene_duration)

                caption_clip = TextClip(
                    scene.image_caption or "",
                    fontsize=self.portrait_caption_font_size,
                    font=self.text_font,
                    color=self.scene_image_text_color,
                    bg_color=self.scene_image_text_bg_color,
                    size=(bg_image.size[0] * 1.2, None),
                    method='caption',
                    align='center'
                ).set_duration(scene_duration).to_RGB()

                spacing = 5
                group_width = max(image_bg_group.w,
caption_clip.w)
                group_height = image_bg_group.h + spacing +
caption_clip.h
                image_pos = ((group_width - image_bg_group.w)
/ 2, 0)
                caption_pos = ((group_width - caption_clip.w)
/ 2, image_bg_group.h + spacing)
                image_caption_group = CompositeVideoClip(
                    [

image_bg_group.set_position(image_pos),
                        caption_clip.set_position(caption_pos)
                    ],
                    size=(group_width, group_height)
                ).set_duration(scene_duration).to_RGB()
```

```
                final_x_img = margin_x + text_with_padding.w -
group_width
                final_y_img = final_y - group_height -
margin_y

                def sliding_position_img(t):
                    slide_duration = 1.0
                    if t < slide_duration:
                        x_val = self.video_width + group_width
- (self.video_width + group_width - final_x_img) * (
                                    t / slide_duration)
                    else:
                        x_val = final_x_img
                    return (x_val, final_y_img)

                image_caption_group =
image_caption_group.set_position(sliding_position_img)
                layers.append(image_caption_group)

            composite_clip =
CompositeVideoClip(layers).set_audio(voiceover_audio)
            return composite_clip

        def generate_video(self, output_path):
            scene_clips = [self._create_scene_clip(scene) for
scene in self.scenes]
            final_video = concatenate_videoclips(scene_clips,
method="compose")
            bg_music =
AudioFileClip(self.background_music_path).volumex(self.backgro
und_volume)
            bg_music = afx.audio_loop(bg_music,
duration=final_video.duration)
            final_audio =
CompositeAudioClip([final_video.audio, bg_music])
            final_video = final_video.set_audio(final_audio)

            if self.logo_path:
                logo = ImageClip(self.logo_path,
transparent=True).set_duration(final_video.duration).resize(

height=self.logo_height).set_position(self.logo_position).set_
opacity(1.0).to_RGB()
                final_video = CompositeVideoClip([final_video,
logo])
```

```
            final_video.write_videofile(output_path,
threads=6,
                                        fps=self.fps,
bitrate=self.bitrate,
                                        codec="libx264",
audio_codec="aac",
                                        logger=None)

        def create_thumbnail(self, video_path, thumbnail_path,
time=1):
            video = VideoFileClip(video_path)
            frame = video.get_frame(time)
            thumbnail = Image.fromarray(frame)  # Convert
numpy array to Image
            thumbnail.save(thumbnail_path)

        def get_video_duration(self, video_path):
            video = VideoFileClip(video_path)
            return int(video.duration)


    def get_video_generator(background_music_path, text_color,
text_bg_color, text_align,
                            scene_image_text_background_color,
scene_image_text_color,
                            quality, orientation,
logo_path=None):
        settings_by_quality = {
            '360p': {
                'video_width': 640,
                'video_height': 360,
                'text_size': 16,
                'text_margin': (10, 10),
                'text_padding': (10, 4),
                'logo_position': (10, 10),
                'logo_height': 70,
                'portrait_height': 70,
                'portrait_caption_font_size': 12,
                'bitrate': "800k",
            },
            '480p': {
                'video_width': 854,
                'video_height': 480,
                'text_size': 21,
                'text_margin': (10, 10),
```

```
                'text_padding': (15, 10),
                'logo_position': (10, 10),
                'logo_height': 70,
                'portrait_height': 70,
                'portrait_caption_font_size': 14,
                'bitrate': "1200k",
            },
            '720p': {
                'video_width': 1280,
                'video_height': 720,
                'text_size': 28,
                'text_margin': (20, 20),
                'text_padding': (20, 10),
                'logo_position': (10, 10),
                'logo_height': 90,
                'portrait_height': 90,
                'portrait_caption_font_size': 18,
                'bitrate': "2500k",
            },
            '1080p': {
                'video_width': 1920,
                'video_height': 1080,
                'text_size': 36,
                'text_margin': (20, 20),
                'text_padding': (25, 15),
                'logo_position': (10, 10),
                'logo_height': 100,
                'portrait_height': 100,
                'portrait_caption_font_size': 24,
                'bitrate': "5000k",
            }
        }

        if quality not in settings_by_quality:
            raise ValueError(f"Unsupported quality
'{quality}'. Choose from: {list(settings_by_quality.keys())}")

        settings = settings_by_quality[quality]

        video_width = settings['video_width']
        video_height = settings['video_height']
        if orientation == 'portrait':
            video_width, video_height = video_height,
video_width

        return VideoGenerator(
```

```
            background_music_path=background_music_path,
            background_volume=0.1,
            text_font='Arial-Bold',
            text_color=text_color,
            text_size=settings['text_size'],
            text_bg_color=text_bg_color,
            text_margin=settings['text_margin'],
            text_padding=settings['text_padding'],
            text_align=text_align,

scene_image_text_bg_color=scene_image_text_background_color,
            scene_image_text_color=scene_image_text_color,
            video_width=video_width,
            video_height=video_height,
            logo_path=logo_path,
            logo_position=settings['logo_position'],
            logo_height=settings['logo_height'],
            portrait_height=settings['portrait_height'],

portrait_caption_font_size=settings['portrait_caption_font_siz
e'],
            bitrate=settings['bitrate']
        )
```

ДОДАТОК Г

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	157 с.
	Інші документи	
2.	Презентаційні матеріали	18 плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Шатило І.Ю.			Автоматизація створення відео на основі короткого змісту тексту з веб-сайтів	Шифр групи	Код напр./спец.
Перевірив		Чала Л.Е.				СШІм-23-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Золотухін О.В.					