ДОДАТОК А

Графічний матеріал атестаційної роботи

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра ЕОМ

Технологія фільтрації контекстно-вбудованої реклами в відеоконтент
Атестаційна робота
Другий (магістерський) рівень

Автор:
Руденко Н.В.,
Ст. гр. КСМзм-19-1

Керівник:
Проф. Рубан І.В.

Харків 2020

1

Мета і задачі роботи

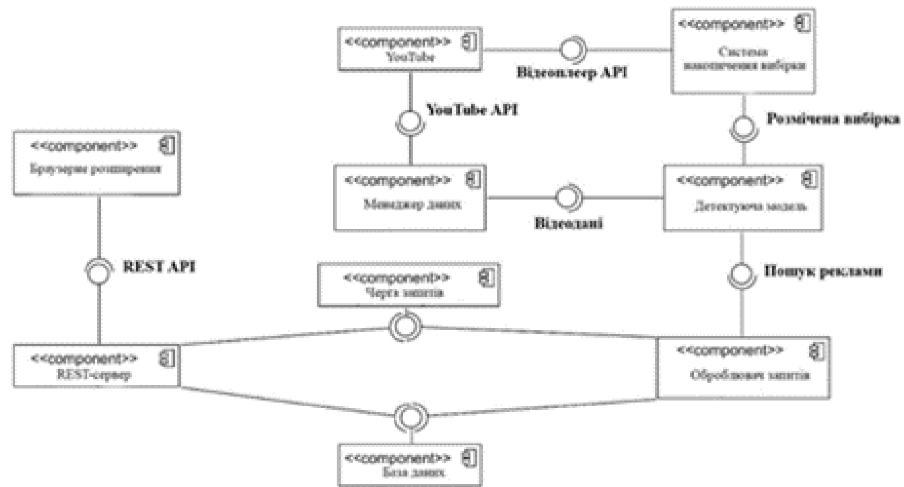
Метою атестаційної роботи є розробка браузерного розширення з функцією інтелектуального фільтра рекламних фрагментів в роликах відеохостингу YouTube.

У ході виконання атестаційної роботи були вирішенні такі завдання:

- вивчення існуючих методів фільтрації реклами в відео;
- розробка системи для ручної розмітки реклами у відеороликах з метою накопичення навчальної вибірки;
- зібрання навчальної вибірки за допомогою розробленої системи;
- розробка та навчання моделі розпізнавання рекламних вставок в відеоролик;
- розробка та протестування REST API модуля фільтрації реклами;
- розробка масштабованої серверної частини для обробки запитів браузерного розширення;
- розробка розширення для браузера.

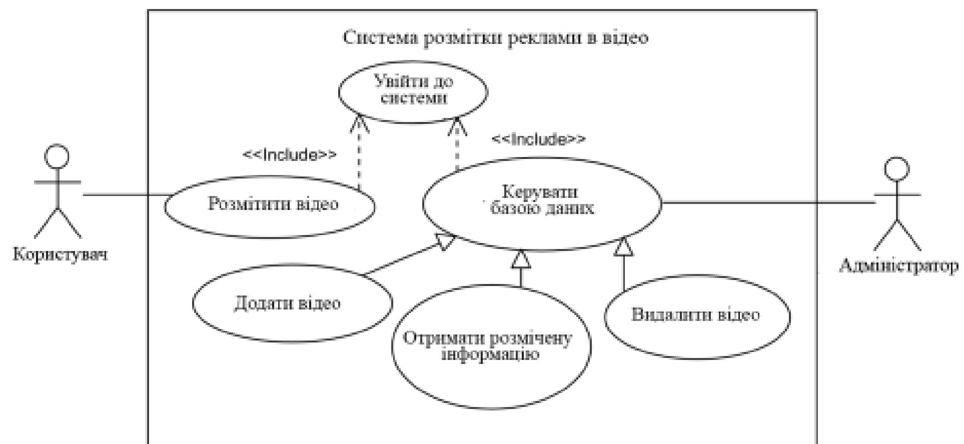
2

Загальна архітектура системи



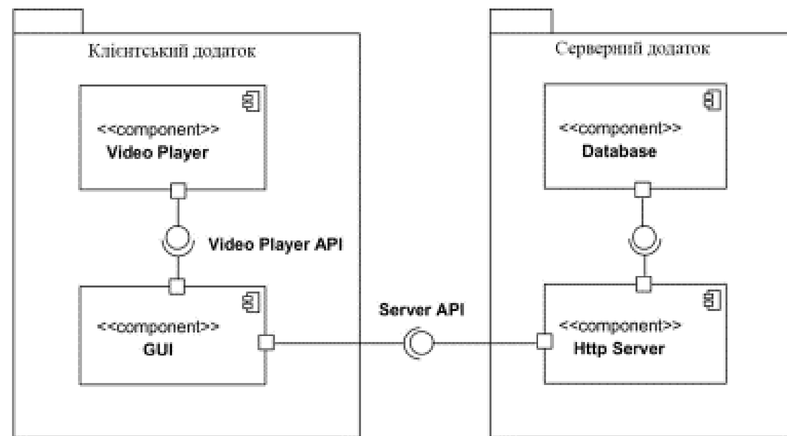
3

Діаграма варіантів використання системи накопичення навчальної вибірки



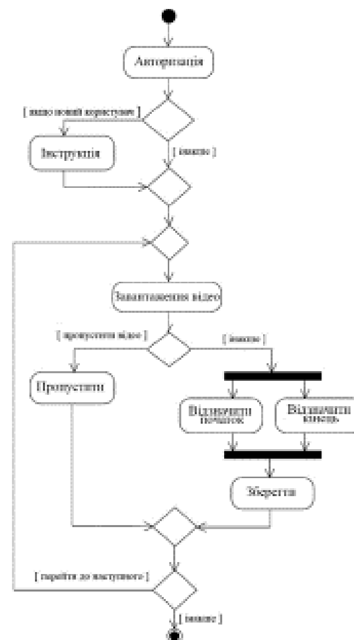
4

Діаграма компонентів системи для накопичення навчальної вибірки



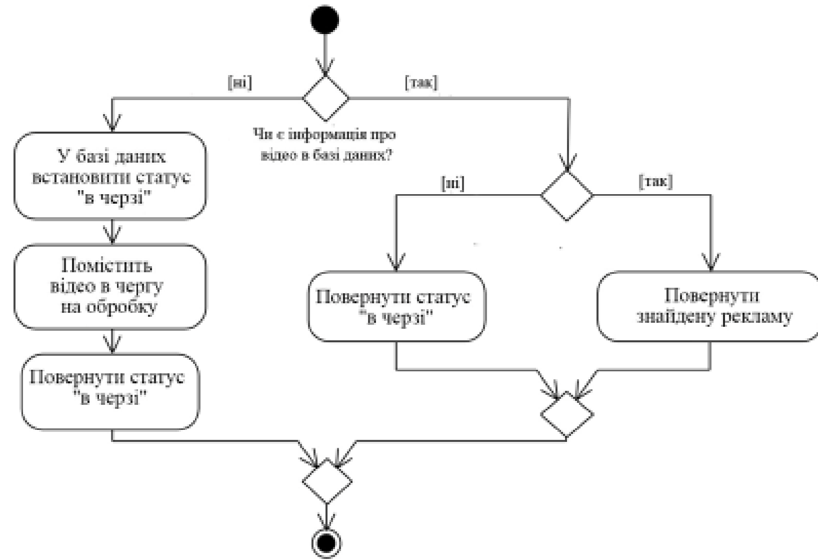
5

Діаграма діяльності користувача в системі для накопичення навчальної вибірки

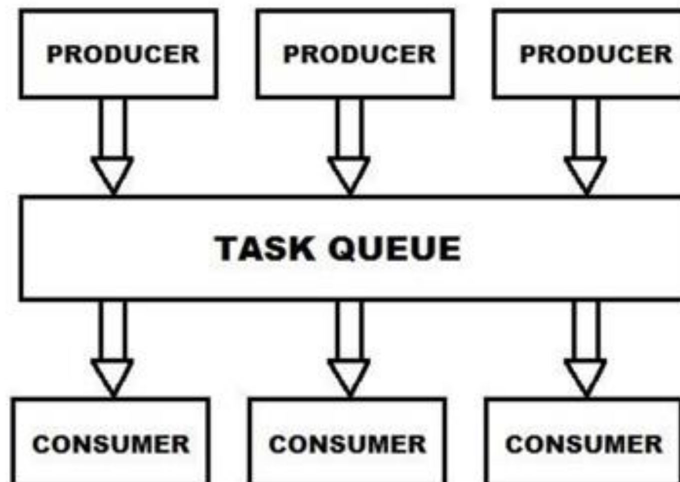


6

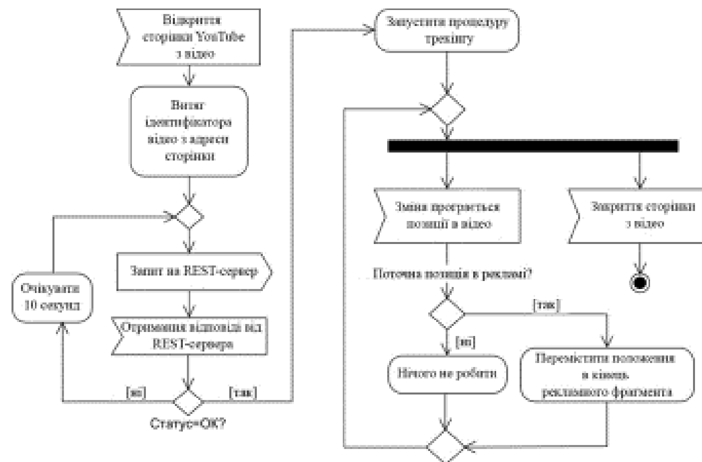
Діаграма діяльності REST-сервера



Архітектура паттерна Producer-Consumer



Діаграма діяльності браузерного розширення



9

Загальна схема роботи детектуючої моделі

10

Висновок

Мета даної роботи полягала в розробці розширення для браузера Google Chrome з функцією інтелектуального фільтрування рекламних фрагментів в роликах відеохостингу YouTube. Мета була досягнута, в ході досягнення цієї мети були вирішені наступні завдання:

- вивчені існуючі методи фільтрації реклами в відео;
- розроблена система для ручної розмітки реклами у відеороликах з метою накопичення навчальної вибірки;
- зібрана навчальна вибірка за допомогою розробленої системи;
- розроблена і навчена модель розпізнавання рекламних вставок в відеоролик;
- розроблений і протестований REST API модуля фільтрації реклами;
- розроблена масштабована серверна частина для обробки запитів браузерного розширення;
- розроблено розширення для браузера.

ДОДАТОК Б

Приклад роботи з програмним комплексом

Розглянемо роботу з програмним комплексом на прикладі. Мета проекту: розробити розширення веб-браузера для виявлення та автоматичного пропуску реклами у відеороликах YouTube, які вставляє блогер на етапі редагування відео.

AdMark – веб-додаток для розмітки реклами у відео для збору навчальних даних.

```
import os
import json
from collections import defaultdict
from argparse import ArgumentParser

from sklearn.model_selection import StratifiedKFold

from AdDetectorModel.models import BuzzwordsCBBasedModel
from AdDetectorModel.utils.evaluation import evaluate
from AdDetectorModel.utils.youtube import YouTubeDownloader
from AdDetectorUtils.paths import *

def main():
    arg_parser = ArgumentParser()
    arg_parser.add_argument('--markups-file', type=str, help='path to file with markups')
    arg_parser.add_argument('--cv-splits', type=int, default=4, help='number of splits in CV')
    args = arg_parser.parse_args()
    if not args.markups_file:
        print('Please specify --markups-file')
        return
    if not os.path.exists(args.markups_file):
        print('Markups file does not exists.', args.markups_file)
        return
    with open(args.markups_file, 'r') as mf:
        markups = json.load(mf)
    video_ids = list(markups.keys())
    ytdl = YouTubeDownloader()
    for i, video_id in enumerate(video_ids):
        print('Preparing video {} ({}/{})'.format(video_id, i+1, len(video_ids)))
        ytdl.load_all(video_id)
    video_ids = list(filter(lambda video_id: subs_exists(video_id) and video_exists(video_id), video_ids))
    print('Total videos: ', len(video_ids))
    has_ads = [len(markups[video_id]) > 0 for video_id in video_ids]
    cv = StratifiedKFold(n_splits=args.cv_splits, shuffle=True, random_state=0)
    avg_metrics = defaultdict(lambda: 0)
```

```
    for split_id, split in enumerate(cv.split(video_ids, has_ads)):
        train_idx, test_idx = split
        print('Cross-validation split #', split_id + 1)
        train_ids = [video_ids[idx] for idx in train_idx]
        test_ids = [video_ids[idx] for idx in test_idx]
        train_markups={video_id:markups[video_id]forvideo_id in 
train_ids}
        test_markups = {video_id: markups[video_id] for video_id in 
test_ids}
        model = BuzzwordsCBBasedModel()
        model.train(train_markups)
        metrics = evaluate(model, test_markups)
        for metric, value in metrics.items():
            avg_metrics[metric] += value
        print(avg_metrics)
    for metric in avg_metrics.keys():
        avg_metrics[metric] /= args.cv_splits
        print('{}: {:.3f}'.format(metric, avg_metrics[metric]))


if __name__ == "__main__":
    main()
```

AdDetectorModel – складається з основної логіки для виявлення реклами.

```
import os

import numpy as np
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
from keras.layers import Dense, LSTM, TimeDistributed, Bidirectional
from keras.models import Sequential, model_from_json

from AdDetectorModel.models import BaseAdDetectorModel
from AdDetectorModel.utils.scenes import SceneDetectionManager
from AdDetectorModel.utils.subtitles import Subtitles
from AdDetectorModel.utils.texts import 
preprocess_russian_text_with_morph
from AdDetectorModel.utils.youtube import VideoInfo


class D2VLSTMBasedModel(BaseAdDetectorModel):
    def __init__(self):
        super().__init__()
        self.sdm    =    SceneDetectionManager(detector_type='content', 
threshold=20, save_scenes=True)
        self.vectorizer = None
        self.subs_part_len = 10
        self.subs_classifier = None
        self._fitted = False
        self.window = 3

    def save(self, path):
        if not self._fitted:
            raise Exception("Model is not fitted yet. Fit model before 
saving.")
        if os.path.exists(path) and not os.path.isdir(path):
            raise Exception("Path for saving should be directory.")
        if not os.path.exists(path):
```

```
                os.mkdir(path)
            with open(os.path.join(path, 'lstm.json'), "w") as f:
                f.write(self.subs_classifier.to_json())
            self.subs_classifier.save_weights(os.path.join(path,
'lstm.weights'))

        def load(self, path):
            with open(os.path.join(path, 'lstm.json'), "w") as f:
                self.subs_classifier = model_from_json(f.read())
            self.subs_classifier.load_weights(os.path.join(path,
'lstm.weights'))
            self._fitted = True

        def find_ads(self, video_ids):
            if not isinstance(video_ids, list):
                video_ids = [video_ids]
            result = []

            for video_id in video_ids:
                info = VideoInfo(video_id)
                subs=Subtitles(video_id, preprocess_russian_text_with_morph)
                scenes = self.sdm.detect_scenes(video_id)
                r = 0
                xs = []
                segments = []
                for l in range(len(scenes)):
                    while  r  +  1  <  len(scenes)  and  scenes[r][1]  -
scenes[l][0] < self.subs_part_len:
                        r += 1
                    words = list(map(str.strip, subs.fulltext(scenes[l][0],
scenes[r][1]).split()))
                    x = list(self.vectorizer.infer_vector(words))
                    x.append((scenes[l][0]+scenes[r][1])/2/info.duration)
                    xs.append(x)
                    segments.append((scenes[l][0], scenes[r][1]))

                segments=[(0,0)]*(self.window//2)+segments+[(0,0)]*
(self.window // 2)
                features = len(xs[0])
                xs  =  [[0]*features]*(self.window  //  2)  +  xs  +  [[0]  *
features] * (self.window // 2)
                X = []
                for  i  in  range(self.window  //  2,  len(xs)  -  self.window  //
2):
                X.append(xs[i - self.window // 2: i + self.window // 2 + 1])
                X = np.array(X)
                Y = self.subs_classifier.predict(X) > 0.5
                ads = []
                for i in range(self.window // 2, len(X) - self.window // 2):
                  cnt = 0
                  for d in range(-(self.window // 2), self.window // 2 + 1):
                        cnt += Y[i + d][self.window // 2 - d][0]
                  if cnt > self.window // 2:
                        ads.append(segments[i])

                merged_ads = []
                for ad in ads:
                  if len(merged_ads) == 0 or ad[0] - merged_ads[-1][1] > 10:
                        merged_ads.append(ad)
                    else:
                        merged_ads.append((merged_ads.pop()[0], ad[1]))
                result.append(merged_ads)

            return result if len(result) > 1 else result[0]
```

```
        def train(self, markups):
            video_ids = list(markups.keys())
            print('start model training')
            tagged_data = []
            subs_parts = []
            Ys = []
            ps = []
            for idx, video_id in enumerate(video_ids):
            print('\rprocessing video {} ({}/{})'.format(video_id, idx + 1,
len(video_ids)), end='')
                subs=Subtitles(video_id, preprocess_russian_text_with_morph)
                scenes = self.sdm.detect_scenes(video_id)
                r = 0
                sp = []
                ys = []
                p = []
                info = VideoInfo(video_id)
                for l in range(len(scenes)):
                    while r + 1 < len(scenes) and scenes[r][1] -
scenes[l][0] < self.subs_part_len:
                        r += 1
                    seg = (scenes[l][0], scenes[r][1])
                    words = list(map(str.strip, subs.fulltext(scenes[l][0],
scenes[r][1]).split()))
                    sp.append(words)
                    tagged_data.append(TaggedDocument(words=words,
tags=[str(len(tagged_data))]))
                    p.append((seg[0] + seg[1]) / 2 / info.duration)
                    if self._inside_ad(seg, markups[video_id]):
                        ys.append(1)
                    elif self._intersect_ad(seg, markups[video_id]):
                        ys.append(0.5)
                    else:
                        ys.append(0)
                subs_parts.append(sp)
                Ys.append(ys)
                ps.append(p)
            self.vectorizer = Doc2Vec(size=50, alpha=0.025, dm=0)
            self.vectorizer.build_vocab(tagged_data)
            self.vectorizer.train(tagged_data,
total_examples=self.vectorizer.corpus_count, epochs=500)
            X = []
            Y = []
            for idx, video_id in enumerate(video_ids):
                print('\rprocessing video {} ({}/{})'.format(video_id, idx +
1, len(video_ids)), end='')
                xs=[self.vectorizer.infer_vector(words)for words in
subs_parts[idx]]
                xs = list(np.hstack((xs, np.reshape(ps[idx], (len(ps[idx]),
1)))).tolist())
                ys = Ys[idx]
                features = len(xs[0])
                xs = [[0] * features] * (self.window // 2) + xs + [[0] *
features] * (self.window // 2)
                ys =[0] * (self.window // 2) + ys + [0] * (self.window // 2)
                for i in range(self.window//2, len(xs) - self.window // 2):
                    X.append(xs[i - self.window//2:i+self.window // 2 + 1])
                    Y.append(ys[i-self.window//2:i + self.window // 2 + 1])

            X = np.array(X)
            Y = np.array(Y)
            Y = Y.reshape((Y.shape[0], Y.shape[1], 1))
            print(X.shape, Y.shape)
```

```
            model = Sequential()
            model.add(Bidirectional(LSTM(100,return_sequences=True),
input_shape=(self.window, X.shape[2])))
            model.add(TimeDistributed(Dense(1, activation='sigmoid')))
            model.compile(optimizer='adam', loss='binary_crossentropy')
            np.random.seed(0)
            model.fit(X, Y, batch_size=len(X), shuffle=True, epochs=500)
            self.subs_classifier = model
            self._fitted = True

        @staticmethod
        def _inside_ad(seg, ads):
            for ad in ads:
                if ad[0] <= seg[0] and seg[1] <= ad[1]:
                    return True
            return False

        @staticmethod
        def _intersect_ad(seg, ads):
            for ad in ads:
                if seg[0] < ad[0] < seg[1] or seg[0] < ad[1] < seg[1]:
                    return True
            return False
```

AdDetectorChromeExtension – розширення браузера для Google Chrome, яке вимагає розташування оголошень із сервера за допомогою ідентифікатора відео та автоматично пропускає ці частини відео.

```
var video = undefined;
var currentVideoId = undefined;
var adsList = [];
var lastUrl = undefined;
const requestInterval = 10000;

setInterval(function() {
    if (document.URL !== lastUrl) {
        lastUrl = document.URL;
        processPage();
    }
}, 1000);

function processPage() {
    console.log("process page");
    video = document.getElementsByTagName('video')[0];
    if (video !== undefined) {
        processVideo();
        video.onloadstart = processVideo;
        video.ontimeupdate = trackVideoAndSkipAds;
    }
}

function processVideo() {
    var newVideoId = extractVideoIdFromUrl();
    if (newVideoId === currentVideoId) return;
    currentVideoId = newVideoId;
    adsList = [];
    if (currentVideoId) {
        console.log("detected page with video " + currentVideoId);
```

```
            loadAds(currentVideoId);
        }
    }

    function trackVideoAndSkipAds() {
        if (!video) return;
        var currentTime = video.currentTime;
        for (var i = 0; i < adsList.length; ++i) {
            var ad = adsList[i];
            var adStart = ad[0], adFinish = ad[1];
            if (adStart < currentTime && currentTime < adFinish) {
                console.log("skipping ad");
                video.currentTime = ad[1];
            }
        }
    }

    function loadAds(videoId) {
        var xhr = new XMLHttpRequest();
        xhr.open('GET', 'http://localhost:1234/' + videoId);
        xhr.onload = function() {
            if (xhr.status !== 200) {
                setTimeout(loadAds.bind(this, videoId), requestInterval);
                return;
            }
            if (videoId !== currentVideoId) return;
            var resp = JSON.parse(xhr.responseText);
            console.log(resp);
            if (resp['status'] === 'OK') {
                adsList = resp['ads'];
            } else if (resp['status'] === 'IN_QUEUE') {
                setTimeout(loadAds.bind(this, videoId), requestInterval);
            }
        };
        xhr.onerror = function () {
            setTimeout(loadAds.bind(this, videoId), requestInterval);
        };
        xhr.send();
    }

    function extractVideoIdFromUrl() {
        return extractUrlParams()['v'];
    }

    function extractUrlParams() {
        var url = document.URL;
        url = url.substr(url.indexOf('?') + 1);
        var args = url.split('&');
        var params = {};
        for (var index in args) {
            var arg = args[index];
            var sepIdx = arg.indexOf('=');
            var argName = arg.substr(0, sepIdx);
            var argValue = arg.substr(sepIdx+1);
            params[argName] = argValue;
        }
        return params;
    }
```

AdDetectorServer - сервер REST, який відповідає на запити з розширення браузера і переводить запити працівникам через чергу повідомлень.

Сервер детектора реклами. REST-сервер відповідає на запити щодо виявлення реклами у відео YouTube.

Кроки для запуску:

- встановіть RabbitMQ;
- встановіть MongoDB;
- вкажіть параметри підключення для RabbitMQ та MongoDB (приклад конфігураційного файлу можна знайти у кореневій частині сховища);
- встановіть змінну середовища CONFIG_PATH з абсолютним шляхом до файлу конфігурації;
- встановіть вимоги за допомогою pip install -r requirements.txt.

```
from flask import Flask, jsonify
from flask_cors import CORS
from pika import BlockingConnection, ConnectionParameters
from pymongo import MongoClient

from AdDetectorUtils.config import config

app = Flask(__name__)
CORS(app)

print('Connecting to RabbitMQ...', end='')
queue_conn = BlockingConnection(ConnectionParameters(
    host=config.get('rabbitmq', 'Host') or 'localhost',
    port=config.get('rabbitmq', 'Port') or 5672))
channel = queue_conn.channel()
channel.queue_declare("queries")
print('OK')

print('Connecting to MongoDB...', end='')
mongo_host = config.get('mongo', 'Host') or 'localhost'
mongo_port = int(config.get('mongo', 'Port')) or 27017
mongo_db_name = config.get('mongo', 'Database') or 'addetector'
mongo_query_results = MongoClient(mongo_host, mongo_port)\
    .get_database(mongo_db_name)\
    .get_collection('query_results')
print('OK')


@app.route('/<video_id>')
def get_ads(video_id):
    obj = mongo_query_results.find_one({'videoId': video_id})
    if obj is None:
        obj = {'videoId': video_id, 'status': 'IN_QUEUE'}
        mongo_query_results.insert_one(obj)
```

```
        channel.basic_publish(exchange='',
                              routing_key='queries',
                              body=video_id)
        result = {'videoId': video_id, 'status': 'IN_QUEUE'}
    else:
        result = {'videoId': video_id, 'status': obj['status']}
        if 'ads' in obj:
            result['ads'] = obj['ads']
    print(result)
    return jsonify(result)


app.run(port=1234)
```

AdDetectorWorker - обробник запитів, він отримує запити від сервера через чергу повідомлень, використовує модель для виявлення оголошень і зберігає результати в базі даних.

```
from argparse import ArgumentParser

from pika import BlockingConnection, ConnectionParameters
from pymongo import MongoClient

from AdDetectorModel.models import BuzzwordsBasedModel
from AdDetectorModel.utils.youtube import YouTubeDownloader
from AdDetectorUtils.config import config
from AdDetectorUtils.paths import subs_exists, video_exists

print('Connecting to MongoDB...', end='')
mongo_host = config.get('mongo', 'Host') or 'localhost'
mongo_port = int(config.get('mongo', 'Port')) or 27017
mongo_db_name = config.get('mongo', 'Database') or 'addetector'
mongo_query_results = MongoClient(mongo_host, mongo_port)\
    .get_database(mongo_db_name)\
    .get_collection('query_results')
print('OK')

model = BuzzwordsBasedModel()

def process_query(ch, method, properties, video_id):
    video_id = video_id.decode(encoding='utf-8')
    print('Processing video {}'.format(video_id))

    ytdl = YouTubeDownloader()
    info = ytdl.load_info(video_id)
    result = {'videoId': video_id}
    if info['is_live']:
        result['status'] = 'LIVE'
    else:
        ytdl.load_subtitles(video_id)
        if not subs_exists(video_id):
            result['status'] = 'NO_SUBS'
        else:
            ytdl.load_video(video_id)
            if not video_exists(video_id):
                result['status'] = 'NO_VIDEO'
            else:
                ads = model.find_ads(video_id)
```

```
                    result['status'] = 'OK'
                    result['ads'] = ads
        print(result)
        mongo_query_results.update_one({'videoId':    video_id},    {'$set':
result})
        ch.basic_ack(delivery_tag=method.delivery_tag)
        print('Finished {}'.format(video_id))

    def main():
        arg_parser = ArgumentParser()
        arg_parser.add_argument('--model-path', type=str)
        args = arg_parser.parse_args()

        print('Loading model...', end='')
        #model.load(args.model_path)
        print('OK')

        print('Connecting to RabbitMQ...', end='')
        queue_conn = BlockingConnection(ConnectionParameters(
            host=config.get('rabbitmq', 'Host') or 'localhost',
            port=config.get('rabbitmq', 'Port') or 5672))
        channel = queue_conn.channel()
        channel.queue_declare("queries")
        channel.basic_consume(queue='queries',
on_message_callback=process_query)
        print('OK')
        channel.start_consuming()

    if __name__ == "__main__":
        main()
```