Корноушкин Илья 6131

Ход выполнения лабораторной работы

В рамках данной лабораторной работы предлагается построить два пайплайна:

Пайплайн, который позволяет получить предсказания для исходных данных с помощью некоторой модели.

Пайплайн, который позволяет обучить или дообучить целевую модель.

Для построения такого пайплайна воспользуемся следующими инструментами:

- Apache Airflow

Первый pipeline

Построенный пайплайн будет выполнять следующие действия поочередно:

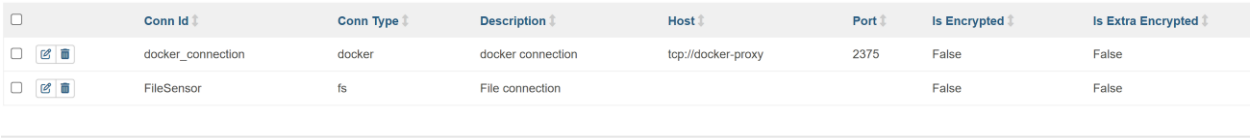
1. Производить мониторинг целевой папки на предмет появления новых видеофайлов.
2. Извлекать аудиодорожку из исходного видеофайла.
3. Преобразовывать аудиодорожку в текст с помощью нейросетевой модели.
4. Формировать конспект на основе полученного текста.
5. Формировать выходной .pdf файл с конспектом.

Для проверки появления новых файлов в целевой папке использовался оператор - FileSensor()

```python
wait_for_new_file = FileSensor(
    task_id='wait_for_new_file',
    poke_interval=10,  # Interval to check for new files (in seconds)
    filepath='/opt/airflow/data',  # Target folder to monitor
    fs_conn_id='FileSensor',
    dag=dag,
)
```

Рисунок 1 – оператор FileSensor

В качестве fs_conn_id использовалось соединение созданное на Airflow по ссылке http://localhost:8080/connection/list/.



Conn Id	Conn Type	Description	Host	Port	Is Encrypted	Is Extra Encrypted
docker_connection	docker	docker connection	tcp://docker-proxy	2375	False	False
FileSensor	fs	File connection			False	False

Рисунок 2 – список использованных соединений

Далее производилось выделение аудиодорожки из входного видео.

```python
extract_audio = DockerOperator(
    task_id='extract_audio',
    image='jrottenberg/ffmpeg',
    command='-i /data/input_video.mp4 -vn -acodec copy /data/audio.aac',
    mounts=[Mount(source='/data', target='/data', type='bind')],
    docker_url="tcp://docker-proxy:2375",
    dag=dag,
)
```

Рисунок 3 – докер оператор для выделения аудиодорожки

Дальше полученную аудиодорожку преобразовываем в текст с помощью нейросетевой модели. Для преобразования аудиозаписи в текст используем следующую модель: https://huggingface.co/openai/whisper-small. Вот какой код приводится для использования модели:

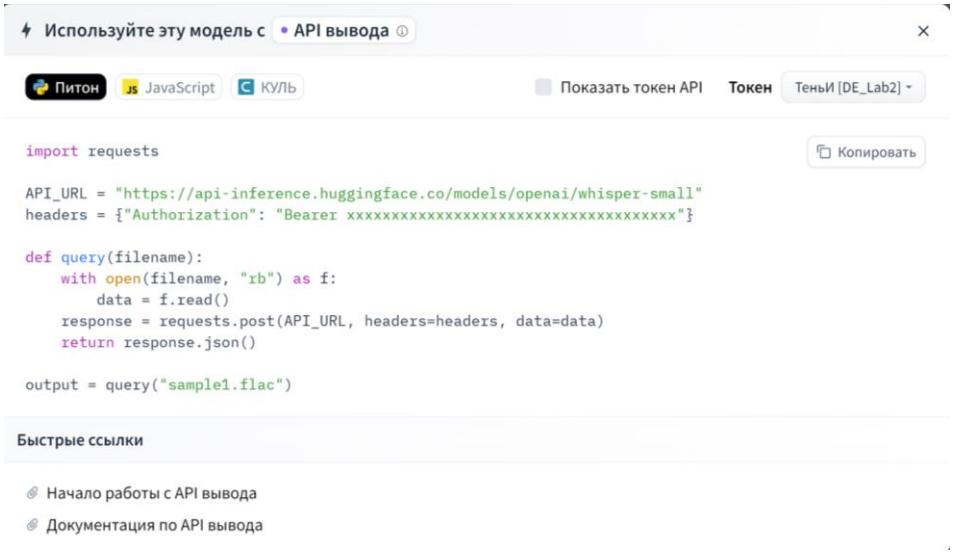


Рисунок 4 – Пример использования нейросетевой модели

Для использования api предварительно был получен токен, который был использован как параметр пост запроса. Также был найден открытый образ с библиотекой requests и создано ещё одно соединение для докера. Для запуска работы используем следующий оператор:

```
transform_audio_to_text = DockerOperator(
    task_id='transform_audio_to_text',
    image='nyurik/alpine-python3-requests',
    command='python /data/audio_to_text.py',
    mounts=[Mount(source='/data', target='/data', type='bind')],
    docker_url="tcp://docker-proxy:2375",
    dag=dag,
)
```

Рисунок 5 – докер оператор для выделения текста

Формируем конспект на основе полученного текста. Для составления конспекта по тексту используем следующую модель: https://huggingface.co/slauw87/bart_summarisation. И применяем следующий оператор:

```python
summarize_text = DockerOperator(
    task_id='summarize_text',
    image='nyurik/alpine-python3-requests',
    command='python /data/text_to_summ.py',
    mounts=[Mount(source='/data', target='/data', type='bind')],
    docker_url="tcp://docker-proxy:2375",
    dag=dag,
)
```

Рисунок 6 – докер оператор для выделения конспекта

Для преобразования конспекта в pdf использовался следующий код:

C: > Users > ilyak > Desktop > DE > Prerequisites > airflow > data > save_to_pdf.py > ...

```python
from fpdf import FPDF

file = open("/data/summ.txt","r")
pdf = FPDF()
pdf.add_page()
for text in file:
    if len(text) <= 20:
        pdf.set_font("Arial","B",size=18) # For title text
        pdf .cell(w=200,h=10, txt=text, Ln=1,align="C")
    else:
        pdf.set_font("Arial",size=15) # For paragraph text
        pdf .multi_cell(w=0,h=10, txt=text,align="L")
    pdf. output ( "/data/output.pdf")
```

Рисунок 7 – код преобразования txt файла в pdf

Использовался следующий оператор. Для использования библиотеки fpdf был найден image.

```python
save_to_pdf = DockerOperator(
    task_id='save_to_pdf',
    image='sasha151299/my_pdf:1.0',
    command='python /data/save_to_pdf.py',
    mounts=[Mount(source='/data', target='/data', type='bind')],
    docker_url="tcp://docker-proxy:2375",
    dag=dag,
)
```

Рисунок 8 – докер оператор для преобразования в pdf

В итоге получаем следующий результат работы:

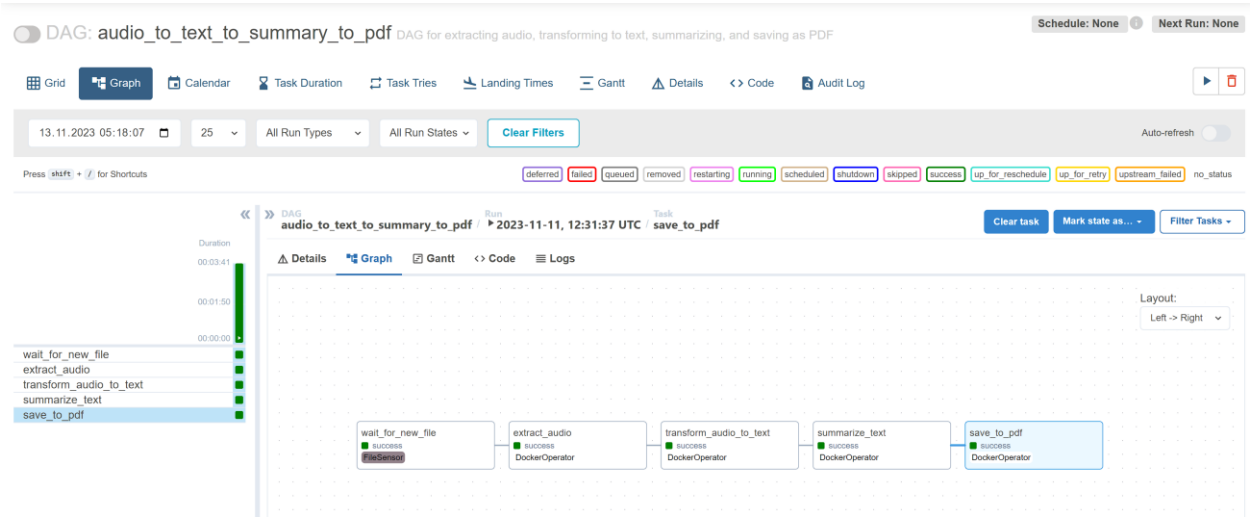


Рисунок 9 – результат запуска DAG

Код DAG представлен в файле – airflow-first-pipeline.

ВТОРОЙ PIPELINE

Построенный пайплайн реализует систему автоматического обучения/дообучения нейросетевой модели.

Для обучения был выбран набор данных MNIST, который будет скачан из tf.keras. В качестве модели была выбрана трёхслойная свёрточная. Сеть состоит из двух свёрточных слоёв и полносвязного слоя. После каждого свёрточного слоя используется батч-номализация, ReLu и dropout слой. После полносвязного слоя идёт softmax.

Итак, пайплайн будет выполнять следующие действия:

1. Читать набор файлов из определенного источника (файловой системы, сетевого интерфейса и т.д.).
2. Формировать пакет данных для обучения модели.
3. Обучать модель.
4. Сохранять данные результатов обучения (логи, значения функции ошибки) в текстовый файл

В итоге получился следующий DAG:

C: > Users > ilyak > Desktop > DE > Prerequisites > airflow > dags > airflow-second-pipeline.py > ...

```python
from datetime import datetime
from airflow import DAG
from docker.types import Mount
from airflow.providers.docker.operators.docker import DockerOperator

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2023, 1, 1),
    'retries': 1,
}

dag = DAG(
    'data_engineering_lab_2',
    default_args=default_args,
    description='DAG for data engineering lab 2: training a neural network',
    schedule_interval=None,
)

read_data = DockerOperator(
    task_id='read_data',
    image='sasha151299/second_pipeline:1.0',
    command='python /data/read_data.py',
    mounts=[Mount(source='/data', target='/data', type='bind')],
    docker_url="tcp://docker-proxy:2375",
    dag=dag,
)

train = DockerOperator(
    task_id='train',
    image='sasha151299/second_pipeline:1.0',
    command='python /data/train.py',
    mounts=[Mount(source='/data', target='/data', type='bind')],
    docker_url="tcp://docker-proxy:2375",
    dag=dag,
)

read_data >> train
```

Рисунок 10 – DAG для второго пайплайна

Для получения данных, создания и обучения модели понадлобилась библиотека tensorflow, pandas, numpy. Был найден image со всеми этими библиотеками. В итоге был произведён запуск DAG.

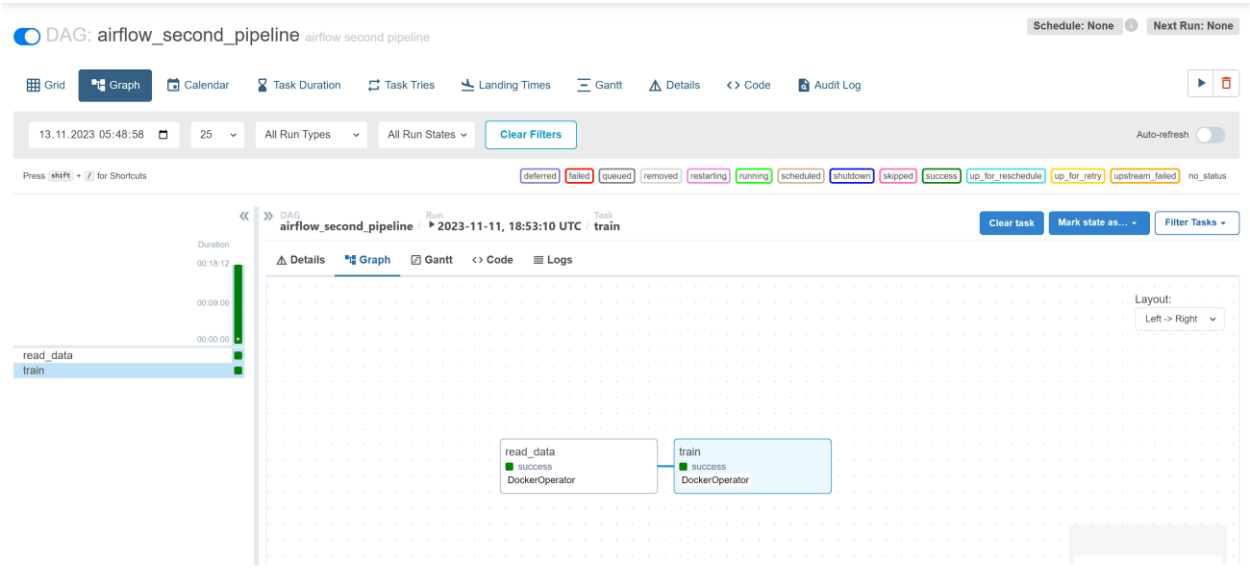


Рисунок 11 – результат запуска DAG

Получены следующие результаты в выходном файле:

```
Iteration 3900, Epoch 6, Loss: 0.035212058573961 26, Accuracy: 98.89965057373047, Val Loss: 0.0756116732954979, Val Accuracy: 97.69999694824219
Iteration 4000, Epoch 6, Loss: 0.033951565623283 86, Accuracy: 98.82127380371094, Val Loss: 0.10865101963281631, Val Accuracy: 97.39999389648438
Iteration 4100, Epoch 6, Loss: 0.03359917551279068, Accuracy: 98.78343963623047, Val Loss: 0.08786778897047043, Val Accuracy: 97.0999984741211
Iteration 4200, Epoch 6, Loss: 0.03213343396782875, Accuracy: 98.85023498535156, Val Loss: 0.09095645695924759, Val Accuracy: 97.0
Iteration 4300, Epoch 6, Loss: 0.03189978748559952, Accuracy: 98.8853530883789, Val Loss: 0.09066455811262131, Val Accuracy: 97.39999389648438
Iteration 4400, Epoch 6, Loss: 0.03176344558596611, Accuracy: 98.88900756835938, Val Loss: 0.10015932470560074, Val Accuracy: 97.29999542236328
Iteration 4500, Epoch 6, Loss: 0.03260624036192894, Accuracy: 98.88692474365234, Val Loss: 0.08204612135887146, Val Accuracy: 97.79999542236328
Iteration 4600, Epoch 7, Loss: 0.02948298119008541, Accuracy: 99.375, Val Loss: 0.09190355241298676, Val Accuracy: 97.0999984741211
Iteration 4700, Epoch 7, Loss: 0.03637147322297096, Accuracy: 98.86904907226562, Val Loss: 0.07542452216148376, Val Accuracy: 97.39999389648438
Iteration 4800, Epoch 7, Loss: 0.033641159534454346, Accuracy: 98.8338394165039, Val Loss: 0.0968259647488594, Val Accuracy: 97.39999389648438
Iteration 4900, Epoch 7, Loss: 0.03184644505381584, Accuracy: 98.8780746459961, Val Loss: 0.11273861676454544, Val Accuracy: 96.4000015258789
Iteration 5000, Epoch 7, Loss: 0.029918067157268524, Accuracy: 98.95061492919922, Val Loss: 0.10414733737707138, Val Accuracy: 97.29999542236328
Iteration 5100, Epoch 7, Loss: 0.02943888120353222, Accuracy: 98.96658325195312, Val Loss: 0.08610555529594421, Val Accuracy: 97.5
Iteration 5200, Epoch 7, Loss: 0.029342522844672203, Accuracy: 98.98243713378906, Val Loss: 0.07723577320575714, Val Accuracy: 97.69999694824219
Iteration 5300, Epoch 7, Loss: 0.028722992166876793, Accuracy: 99.00487518310547, Val Loss: 0.09612444788217545, Val Accuracy: 97.29999542236328
Iteration 5400, Epoch 8, Loss: 0.022173220291733742, Accuracy: 99.23878479003906, Val Loss: 0.07811415195465088, Val Accuracy: 97.5999984741211
Iteration 5500, Epoch 8, Loss: 0.026820266619324684, Accuracy: 99.10072326660156, Val Loss: 0.07147786021232605, Val Accuracy: 97.89999389648438
Iteration 5600, Epoch 8, Loss: 0.025247029960155487, Accuracy: 99.1304931640625, Val Loss: 0.09150978922843933, Val Accuracy: 97.0999984741211
Iteration 5700, Epoch 8, Loss: 0.023780878633260727, Accuracy: 99.17957305908203, Val Loss: 0.08728518337011337, Val Accuracy: 97.39999389648438
Iteration 5800, Epoch 8, Loss: 0.02348487451672554, Accuracy: 99.18849182128906, Val Loss: 0.10284123569726944, Val Accuracy: 97.0999984741211
Iteration 5900, Epoch 8, Loss: 0.022813649848103523, Accuracy: 99.23469543457031, Val Loss: 0.09645788371562958, Val Accuracy: 96.80000305175781
Iteration 6000, Epoch 8, Loss: 0.022999532520771027, Accuracy: 99.21263122558594, Val Loss: 0.09392204135656357, Val Accuracy: 97.19999694824219
Iteration 6100, Epoch 8, Loss: 0.023082105442881584, Accuracy: 99.20712280273438, Val Loss: 0.11645585298538208, Val Accuracy: 97.29999542236328
Iteration 6200, Epoch 9, Loss: 0.021723456680774 69, Accuracy: 99.18663787841797, Val Loss: 0.09199878573417664, Val Accuracy: 97.19999694824219
Iteration 6300, Epoch 9, Loss: 0.022282099351286888, Accuracy: 99.2142333984375, Val Loss: 0.12165850400924683, Val Accuracy: 97.0
Iteration 6400, Epoch 9, Loss: 0.021440500393509865, Accuracy: 99.22733306884766, Val Loss: 0.08671515434980392, Val Accuracy: 97.5999984741211
Iteration 6500, Epoch 9, Loss: 0.021748295053839684, Accuracy: 99.19570922851562, Val Loss: 0.08911909908056259, Val Accuracy: 97.5
Iteration 6600, Epoch 9, Loss: 0.021629042923450 47, Accuracy: 99.22701263427734, Val Loss: 0.09445776790380478, Val Accuracy: 97.39999389648438
Iteration 6700, Epoch 9, Loss: 0.021213021129369736, Accuracy: 99.23919677734375, Val Loss: 0.11364275962114334, Val Accuracy: 97.39999389648438
Iteration 6800, Epoch 9, Loss: 0.021368181332945824, Accuracy: 99.2570571899414, Val Loss: 0.09559743106365204, Val Accuracy: 97.5999984741211
Iteration 6900, Epoch 10, Loss: 0.024991085752844 81, Accuracy: 99.33036041259766, Val Loss: 0.08460661023885 21, Val Accuracy: 97.29999542236328
Iteration 7000, Epoch 10, Loss: 0.018147772178053856, Accuracy: 99.43048858642578, Val Loss: 0.08454252034425735, Val Accuracy: 97.39999389648438
Iteration 7100, Epoch 10, Loss: 0.019723141565918922, Accuracy: 99.35839080810547, Val Loss: 0.09663759171962738, Val Accuracy: 97.79999542236328
Iteration 7200, Epoch 10, Loss: 0.019628439098596573, Accuracy: 99.32308959960938, Val Loss: 0.10260256379842758, Val Accuracy: 97.29999542236328
Iteration 7300, Epoch 10, Loss: 0.018959112465381622, Accuracy: 99.35503387451172, Val Loss: 0.08630099147558212, Val Accuracy: 97.19999694824219
Iteration 7400, Epoch 10, Loss: 0.019127344712615013, Accuracy: 99.3404769897461, Val Loss: 0.1103690043091774, Val Accuracy: 97.19999694824219
Iteration 7500, Epoch 10, Loss: 0.019469255581498146, Accuracy: 99.31012725830078, Val Loss: 0.1046498566865921, Val Accuracy: 96.69999694824219
Iteration 7600, Epoch 10, Loss: 0.019412869587540627, Accuracy: 99.31488800048828, Val Loss: 0.10169955343008041, Val Accuracy: 97.5999984741211
```

Рисунок 12 – история обучения модели