MLOps Home Project

Alec Gheeraert

MCT-AI

Explanation

Target For my home project i have created an AI model that predicts emotions based on images. In the fictional company this API will be used to ensure customer satisfaction and safety. The target demographics for this API are bars and clubs. For this API they can upload security cameras images and get back what emotions those customers are experiencing. This information can then be used by the bars to adapt to these emotions by changing music. This information can also be applied to ensure the safety for everyone and prevent possible fight or discussions between drunk customers.

Data The API takes in any image, these will then be processed to a 48 by 48 pixel size to be lightweight for prediction. The dataset that was used for training consists of 800 images of happy, sad, angry and fearful facial expressions.

Model The model created is a convolutional neural network. It has 6 hidden layers and uses a rectified linear unit activation function. A softmax output layer is also used.

Support

Architectures This workflow and API has been created to run and be supported on X64 and ARM64. The docker container is also compiled including X64 and ARM64 binaries.

Preparation

Load The data used to train the model is stored in a azure storage account. In here is container called emotions. Here you can upload new training images in their corresponding folders. These folders are each linked to a dataset inside a datastore.

Process To process the new images you have to set pipeline_process to true. This will then download all the images for each emotion. Then each images will be processed and resized to a 48 by 48 pixel size. Afterwards they are uploaded

to a new dataset to seperate the processed from raw images. Each dataset also gets a git-sha for workflow identification.

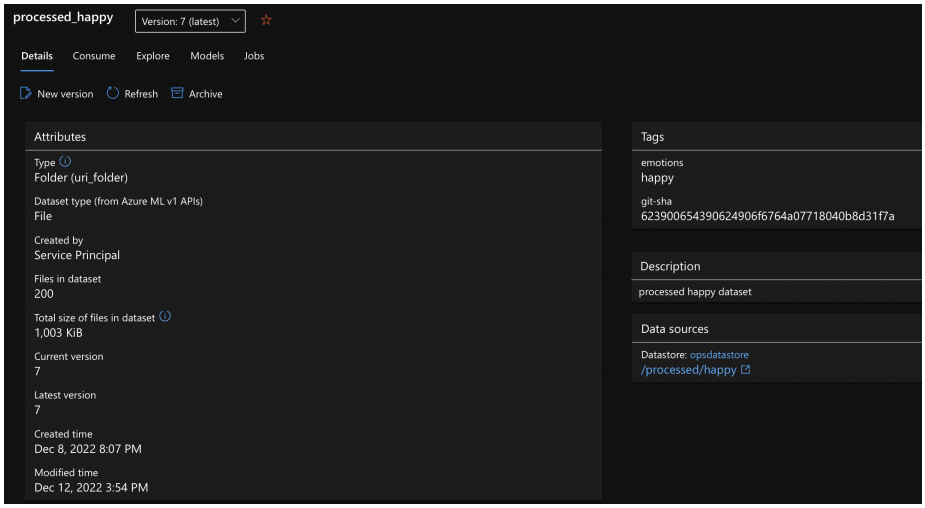


Figure 1: processed happy dataset

Split The last step in the data preparation is to split the processed datasets into a training and testing dataset. This ratio is configured from a variable inside the GitHub workflow. The default that was used for this model is 0.2, meaning 80% of images is used for training and 20% for testing. This setting also has to be set to true to run.

Training

Compute To train the model a azure compute cluster is used which can autmatically scales up to 4 nodes.

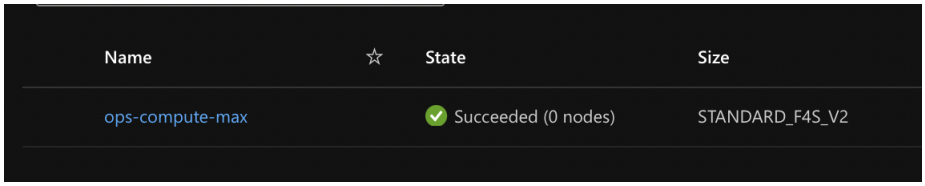


Figure 2: compute cluster

Job Each training request creates a new job which after running the scripts save the newly trained model. These jobs contains the same git-sha as the

datasets do.

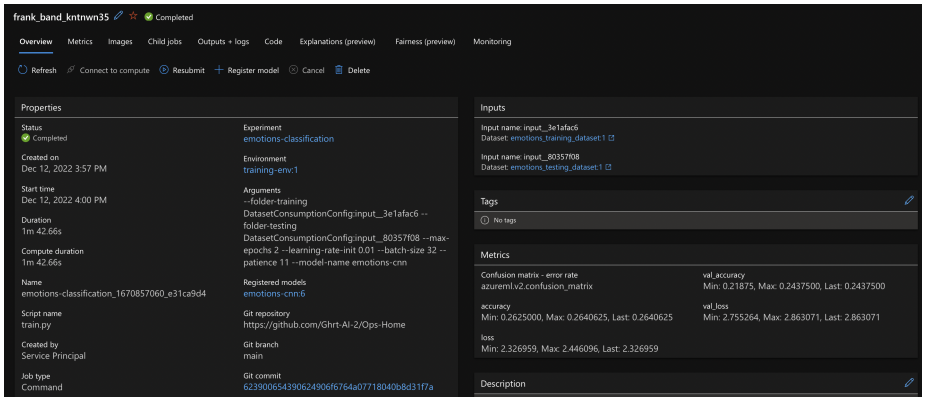


Figure 3: training job details

Logging Each job contains logs and metrics to see any errors and statistics on how the model preformed.

Figure 4: training job metrics

Model After a job is succesfull they are registered and can be accessed as a model. These models are then downloaded to be used in the API.

Deployment

Fastapi To predict images and provide an endpoint for the users of the API. A Fastapi has been created. This takes in any image and resizes it to 48 by 48 pixels to be put into the latest model. This in turn returns a emotion which can then be used by the user however they see best fit.

Docker & GitHub The Fastapi is then dockerized to be easily accessible and scalable. This docker container is uploaded and stored in GitHub Container

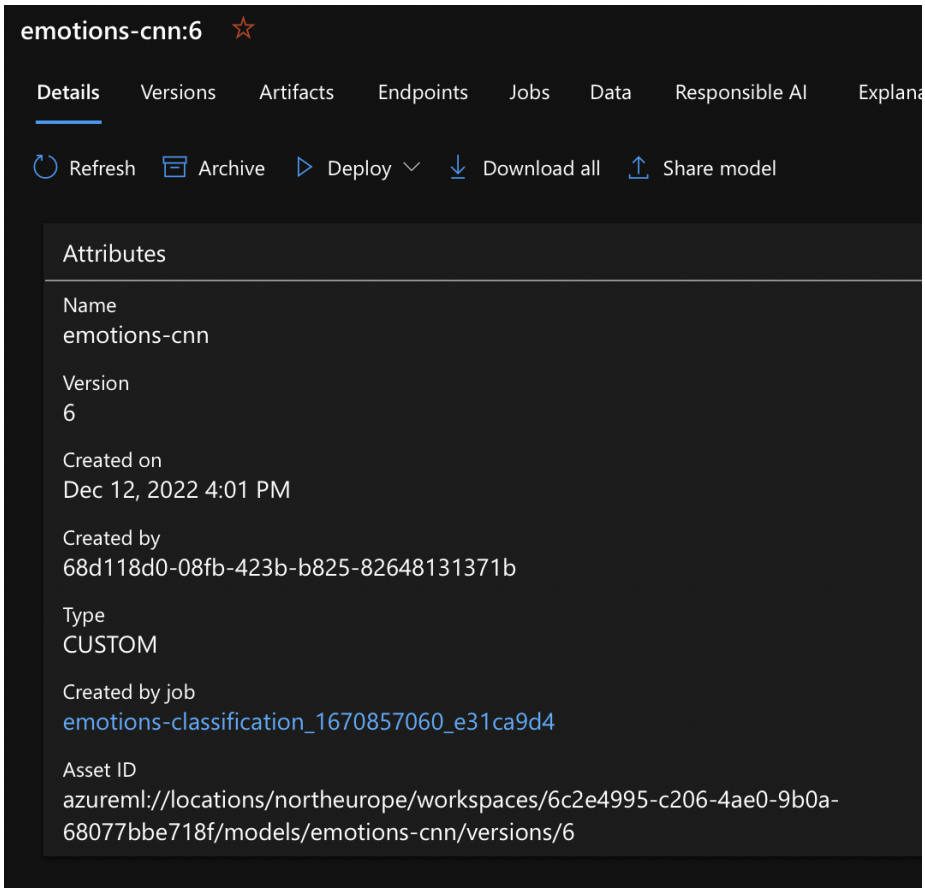


Figure 5: training model

Registry. Each container also gets the same git-sha as in the preparation and training step. This way any problem can be traced back to their origin.

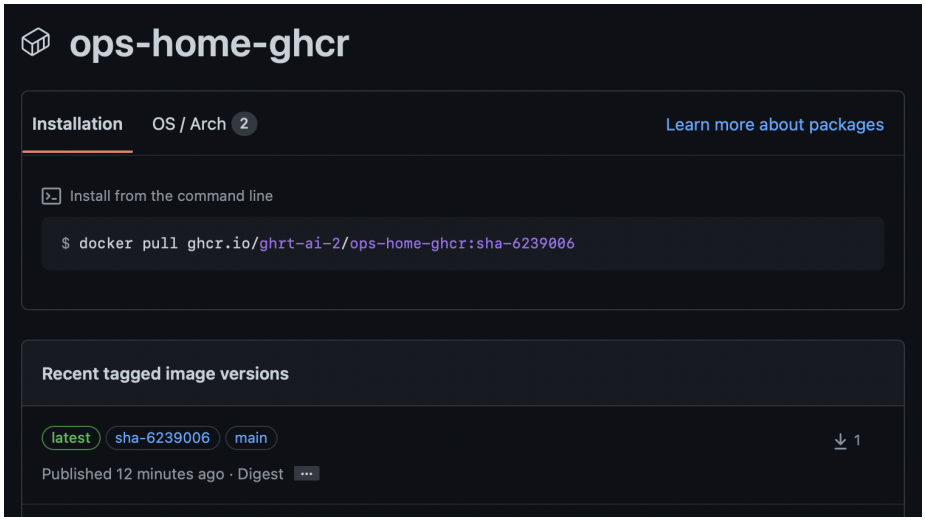


Figure 6: GitHub Container Registry

Kubernetes

Disclamers A GitHub runner has to be running. A local kubernetes version has to be running.

Preparations To be able to always use the latest docker image in our kubernetes cluster. A pull has to be done with a specific tag/version. In this case the git-sha. This is done by editing the kubernetes/deployment.yml file. Here the tag gets replaced by the git-sha used in the previous steps. Therefore using the latest version.

Deployment Once the GitHub workflow has finished the config files are automatically applied with kubectl and will start in kubernetes.

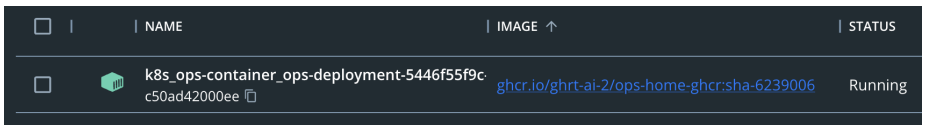
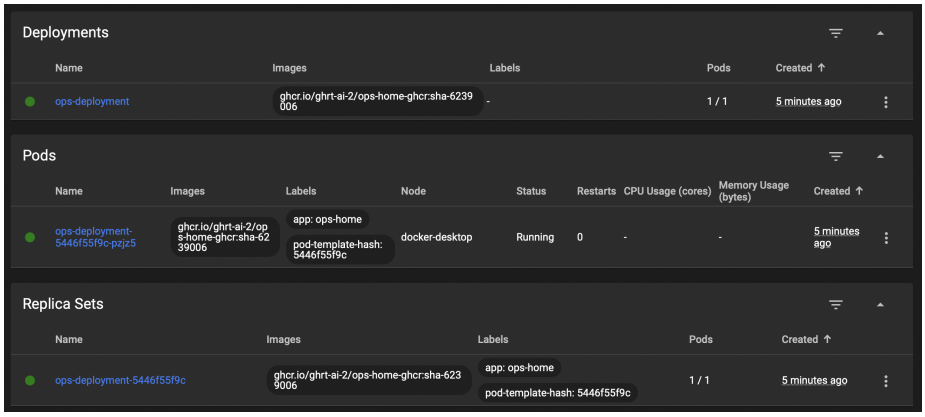


Figure 7: GitHub Actions



Once the deployment and service are running. The command for port forwarding has to be ran. This way the container and therefore API can be accessed through localhost.

```
kubectl apply -f kubernetes/deployment.yml
kubectl apply -f kubernetes/service.yml
kubectl rollout status deployment ops-deployment -n ops-home --timeout=600s
kubectl expose deployment ops-deployment --type=LoadBalancer --port=80 --target-port=8000
```

Figure 8: expose

Usage

Localhost Once the previous steps have been completed the API is accessible on localhost. Then a image can be uploaded and a prediction returned.

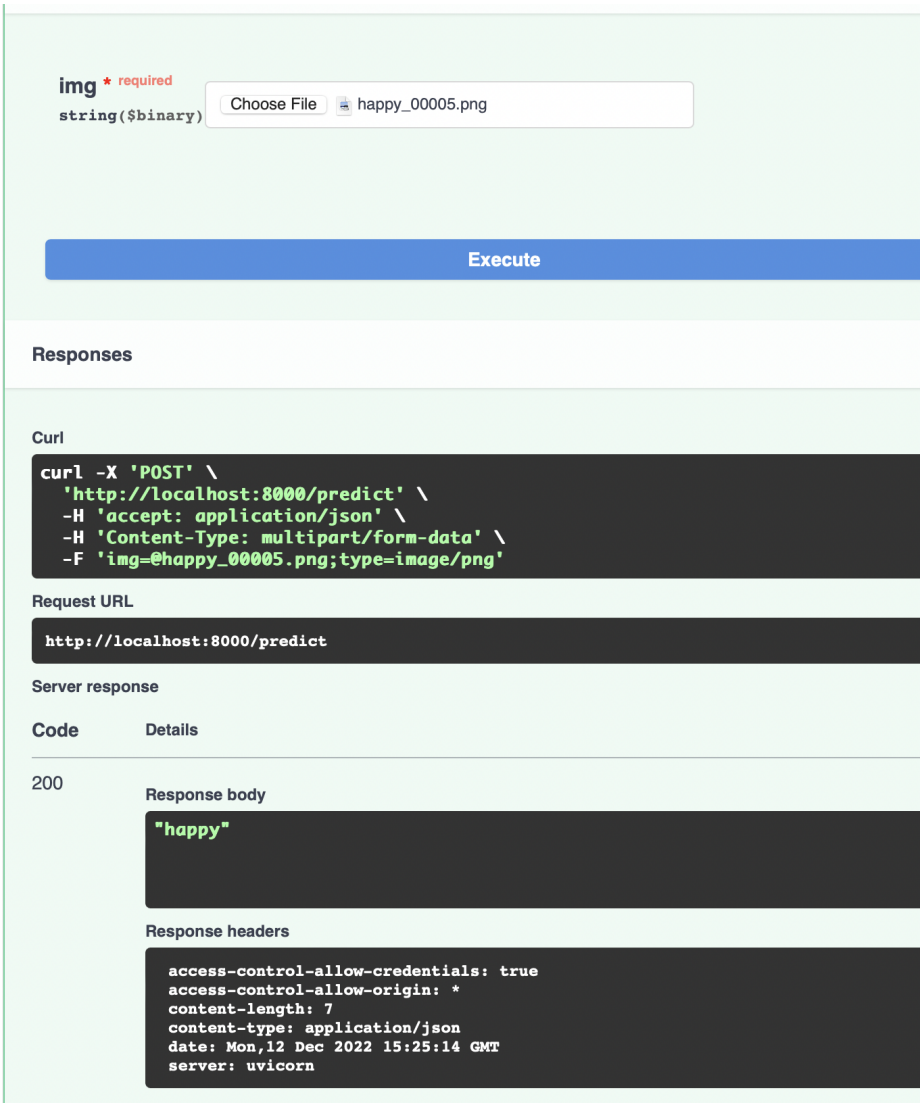
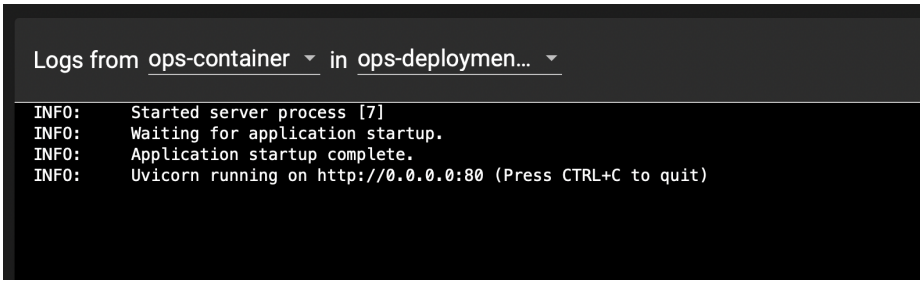


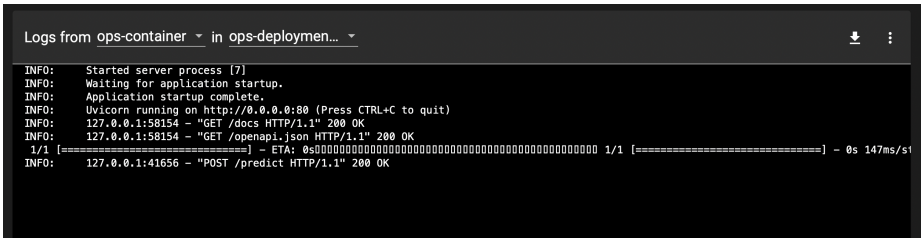
Figure 9: API on localhost

Logs The status of the API can be seen in the logs on kubernetes.

Before request



After request



Automation

To automate all these tasks, a GitHub Actions Workflow has been created.

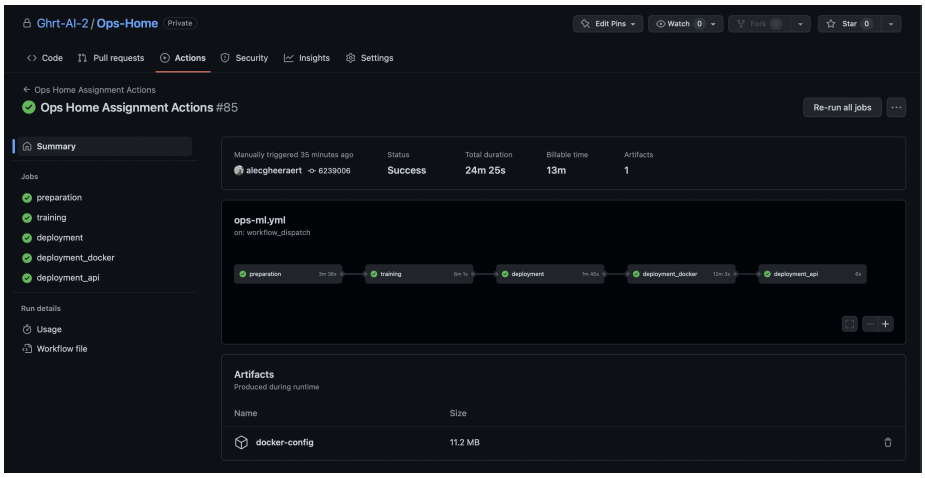


Figure 10: GitHub workflow

Dataset

Original dataset