

MLOps Project Report

(Side note: I'm sorry if I went a little overboard with the project and the documentation.... I had a bit too much fun adding stuff once I got the hang of it.)

Table of Contents

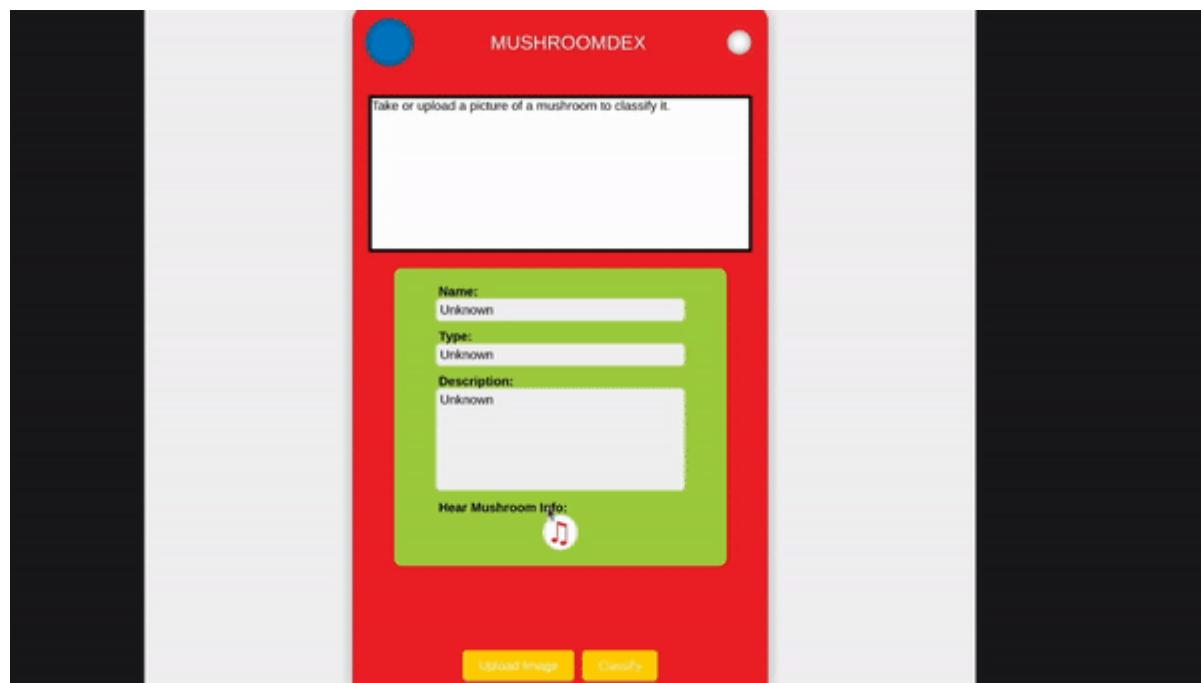
- MLOps Project Report
 - Table of Contents
 - 1. Introduction
 - 2. Project Overview
 - 2.1 Dataset
 - 2.1.1 Data upload
 - 2.2 AI Model Selection
 - 2.2.1 Model decision
 - 2.3 Data Preprocessing
 - 2.3.1 Data preparation for this specific dataset (done outside of the pipeline)
 - 2.3.2 Data prep before training
 - 2.3.3 Target Label Extraction
 - 2.3.4 Label Encoding
 - 2.3.5 Feature Extraction
 - Prerequisites
 - Azure Credentials
 - GitHub Secrets
 - Getting Started
 - 3. Cloud AI Services
 - 3.1 Compute Resource Management
 - 3.2 Environment Setup
 - 4. Model Training and Evaluation
 - 4.1 Azure ML
 - 4.2 Model Training
 - 4.3 Model Evaluation
 - 4.3.1 conclusions
 - 5. Deployment
 - 5.1 Fastapi Deployment
 - 5.3 API Endpoints
 - 5.4 Web App
 - 5.6 Gradio
 - 5.7 Deployment on azure kubernetes !!! (bonus)
 - 6. Integration with Existing Software
 - 6.1 Fake company
 - 6.1.1 Integration in an existing software system
 - 7. Automation and Version Control
 - 7.1 GitHub Actions
 - Jobs
 - All this can then be tested (locally) with the test.yaml file
 - 7.2 Version Control
 - 7.2.1 Training model version
 - 7.2.2 Model Version Retrieval
 - 7.2.3 Model Artifact Storage
 - 7.2.4 Model Deployment Version
 - 8. Conclusion
 - 9. Useful links

- Demos
- Download model (optional)

1. Introduction

This report outlines the details of my MLOps project of mushroom classification. The project's primary objective is to demonstrate the principles of MLOps by building an end-to-end pipeline for creating, training, deploying, and testing a machine learning model using Azure Machine Learning (Azure ML) services. The report highlights key aspects of the project, including data preparation, model training and evaluation, deployment with FastAPI, integration possibilities, and automation strategies.

[Look at demo videos](#)



2. Project Overview

2.1 Dataset

The project involves the classification of mushrooms into nine distinct categories. The dataset used for this task contains of 6000+ images of 9 different mushroom families, each separated in its own folder.

The dataset I used: <https://www.kaggle.com/datasets/lizhecheng/mushroom-classification>

Mushroom Classification Dataset

Image Classification



[Data Card](#) [Code \(4\)](#) [Discussion \(2\)](#)

About Dataset

This dataset is used to classify 9 kinds of mushroom.

If you like this dataset and find it useful, please give a thumbs up    



Pay attention that there are some truncated images in this dataset(maybe only one, I'm not sure), so you are not recommended to use function like ImageFolder to load the whole dataset.

Otherwise, you will get OS Error, which I have shown you in my notebook. It's better to rewrite Dataset class.

Usability

7.50

License

[Community Data License Agree...](#)

Expected update frequency

Never

Tags

Online Communities

Biology

Computer Vision

Image Classification

The dataset contains images like this:



2.1.1 Data upload

I manually uploaded the folders onto azure ml data assets for this project.

A screenshot of the Azure AI | Machine Learning Studio interface, specifically the Data assets section. The URL in the address bar is https://ml.azure.com/data?ws=lucasmlps. The page shows a list of data assets created from local files in the workspace. The data assets listed are: suillus, russula, lactarius, hygrocybe, entoloma, cortinarius, boletus, amanita, and agaricus. All assets are from the current workspace and are at version 1.

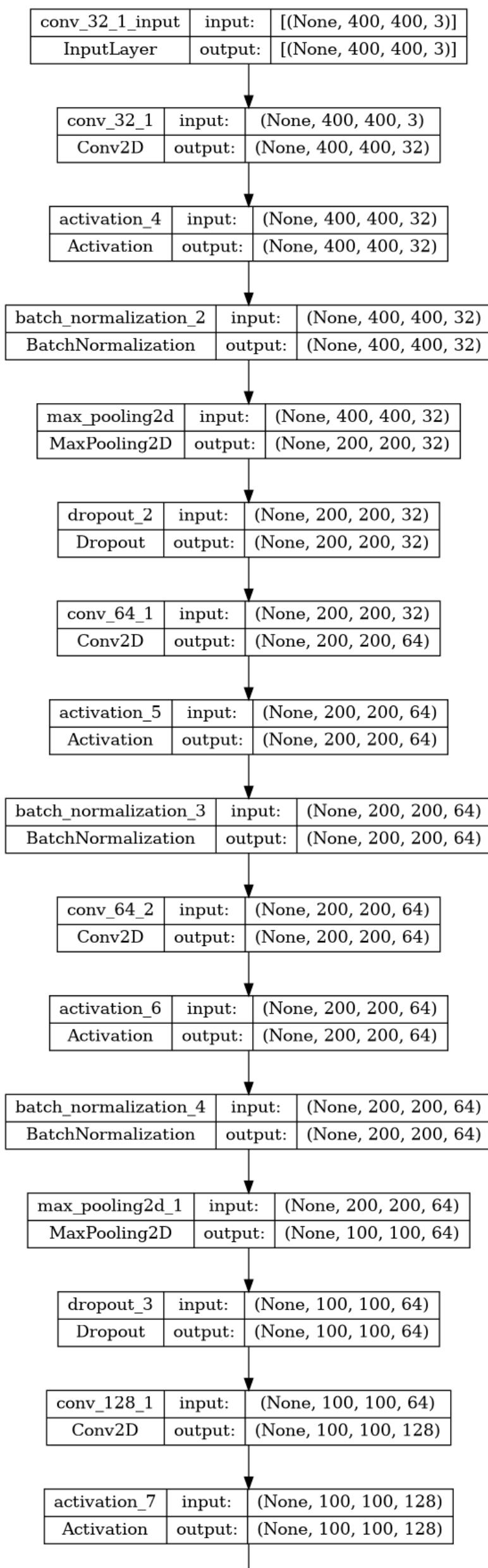
Name	Source	Version
suillus	This workspace	1
russula	This workspace	1
lactarius	This workspace	1
hygrocybe	This workspace	1
entoloma	This workspace	1
cortinarius	This workspace	1
boletus	This workspace	1
amanita	This workspace	1
agaricus	This workspace	1

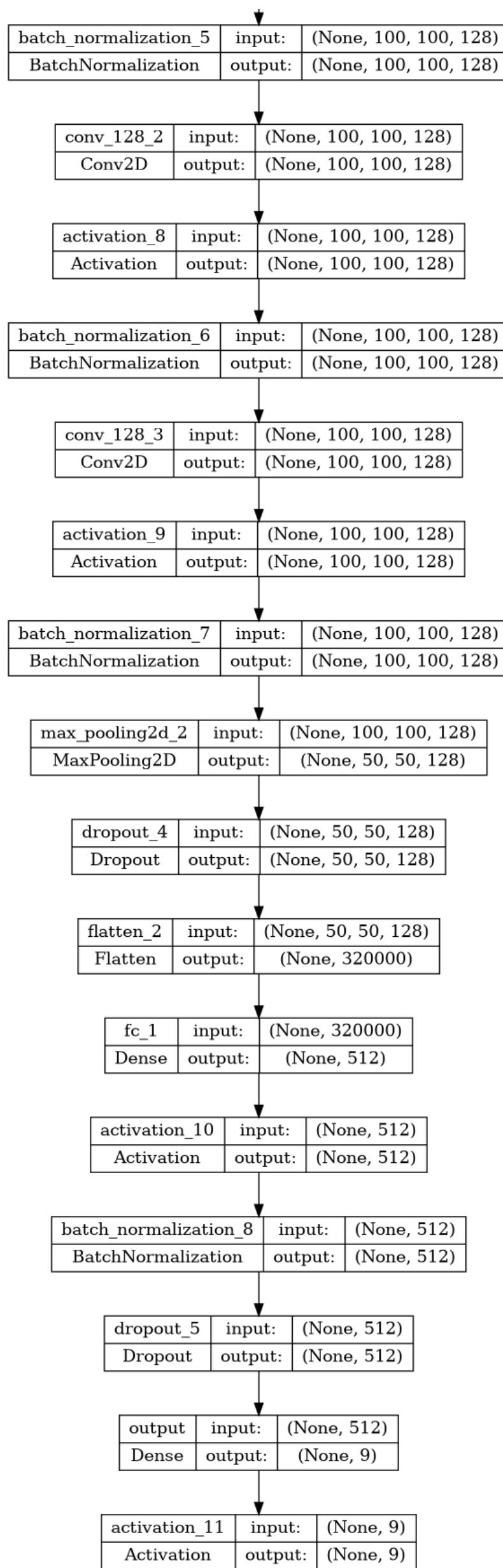
2.2 AI Model Selection

The chosen model for this mushroom classification project is based on the VGG19 architecture with transfer learning. It performed better than a custom CNN.

2.2.1 Model decision

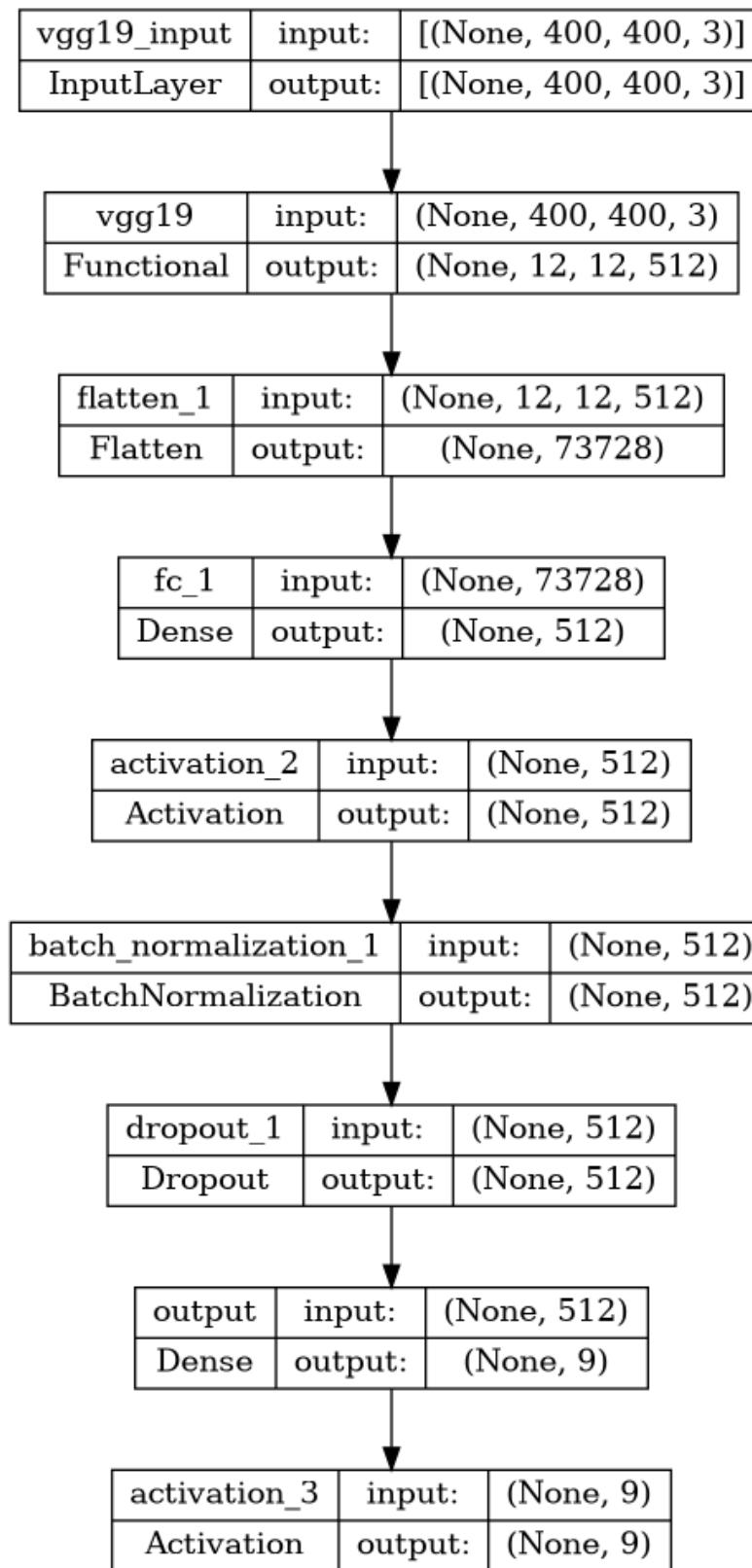
Custom CNN architecture:





Result		precision	recall	f1-score	support
	0	0.20	0.03	0.05	71
	1	0.62	0.10	0.17	150
	2	0.46	0.67	0.55	215
	3	0.31	0.10	0.15	168
	4	0.38	0.04	0.07	73
	5	0.50	0.05	0.09	64
	6	0.34	0.73	0.46	313
	7	0.31	0.33	0.32	151
	8	0.08	0.02	0.03	63
	accuracy			0.37	1268
	macro avg	0.35	0.23	0.21	1268
	weighted avg	0.37	0.37	0.30	1268
	[[2 1 5 1 0 1 50 11 0] [2 15 38 5 0 1 53 35 1] [0 0 144 3 0 1 52 13 2] [1 1 32 16 0 0 100 15 3] [1 1 5 6 3 0 49 7 1] [0 3 22 3 0 3 25 6 2] [4 3 33 13 5 0 230 22 3] [0 0 14 1 0 0 86 50 0] [0 0 19 4 0 0 36 3 1]]				
•	DONE TRAINING				

Model architecture with Transfer learning (VGG19 imagenet):



Results with transfer learning (VGG19):

		precision	recall	f1-score	support
	0	0.51	0.55	0.53	71
	1	0.79	0.81	0.80	150
	2	0.79	0.90	0.84	215
	3	0.72	0.66	0.69	168
	4	0.71	0.53	0.61	73
	5	0.70	0.66	0.68	64
	6	0.72	0.71	0.71	313
	7	0.61	0.66	0.63	151
	8	0.73	0.57	0.64	63
	accuracy			0.71	1268
	macro avg	0.70	0.67	0.68	1268
	weighted avg	0.71	0.71	0.71	1268
[[39 7 4 1 2 0 12 5 1] [7 122 6 2 0 0 5 8 0] [2 4 193 3 1 0 7 2 3] [5 6 10 111 6 3 16 7 4] [3 1 2 5 39 7 8 8 0] [0 1 1 7 3 42 4 6 0] [13 7 11 18 4 4 223 28 5] [5 4 7 2 0 4 30 99 0] [2 2 11 6 0 0 6 0 36]]					

The transfer learning model had better accuracy and recall on all the different classes, so I decided to use that one.

2.3 Data Preprocessing

Effective data preprocessing is a really important step in machine learning projects, ensuring that the dataset is ready for model training. Here are the steps I took to prepare the data:

2.3.1 Data preparation for this specific dataset (done outside of the pipeline)

```
# assuming the dataset is in the ./Mushrooms directory
#for ./Mushrooms/Agaricus rename all images to their 'className + _ + number.jpg'
import os
import sys
import shutil

path = './Mushrooms/'

for x in sorted(os.listdir(path)):
    print(x)
    index_y = 0
    for y in sorted(os.listdir(path + x)):
        #split y on _ and take the second part
        name = y.split('_')[1]
        nr = y.split('_')[0]
        #remove .jpg
        name = name.split('.')[0]
        #add class name
        name = x+'_'+nr+'.jpg'

        #rename
        os.rename(path+x+'/'+y, path+x+'/'+name)
```

2.3.2 Data prep before training

this resizes the images to 400x400, and saves them in the output directory.

```
output_dir = args.output_data
size = (400, 400) # Later we can also pass this as a property

for file in glob(args.data + "/*.jpg"):
    try:
        img = Image.open(file)
        img_resized = img.resize(size)

        # Save the resized image with the new name to the output directory
        output_file = os.path.join(output_dir, os.path.basename(file))
        img_resized.save(output_file)

    except OSError as e:
        print(f"Error processing {file}: {e}")
```

then there is traintestsplit.py, this just takes some images and puts them in a training_folder and test_folder. this is really basic so I wont go into detail here.

2.3.3 Target Label Extraction

This function extracts the mushroom category (label) from the file paths, providing the ground truth labels for each image.

```
def getTargets(filepaths: List[str]) -> List[str]:
    labels = [fp.split('/')[-1].split('_')[0] for fp in filepaths]
    return labels
```

2.3.4 Label Encoding

The LabelEncoder maps each unique mushroom category to a numerical value and then converts these numerical labels into one-hot encoded vectors, which are compatible with machine learning models.

```
def encodeLabels(y_train: List, y_test: List):
    label_encoder = LabelEncoder()
    y_train_labels = label_encoder.fit_transform(y_train)
    y_test_labels = label_encoder.transform(y_test)

    y_train_1h = to_categorical(y_train_labels)
    y_test_1h = to_categorical(y_test_labels)

    LABELS = label_encoder.classes_
    print(f"{LABELS} -- {label_encoder.transform(LABELS)}")

    return LABELS, y_train_1h, y_test_1h
```

2.3.5 Feature Extraction

This function reads and converts the images into numerical arrays (pixel values), preparing them for training.

```
def getFeatures(filepaths: List[str]) -> np.array:
    images = []
    for imagePath in filepaths:
        image = Image.open(imagePath).convert("RGB")
        image = np.array(image)
        images.append(image)
    return np.array(images)
```

Prerequisites

Before continuing, ensure you have the following prerequisites set up:

Azure Credentials

Azure subscription with necessary permissions to create and manage Azure ML resources. Service principal with contributor access to the Azure subscription.

Create a service principal auth token with the Azure CLI:

```
az ad sp create-for-rbac --name "<NAME>" --role contributor --scopes /subscriptions/<SUBSCRIPTION_ID> --json-auth
```

It will look something like this:

```
{
    "clientId": "<CLIENT_ID>",
    "clientSecret": "<CLIENT_SECRET>",
    "subscriptionId": "<SUBSCRIPTION_ID>","
```

```
        "tenantId": "<TENANT_ID>"  
    }
```

GitHub Secrets

The following secrets need to be configured in your GitHub repository for the pipelines to authenticate and interact with Azure resources:

AZURE_CREDENTIALS: A JSON string containing your Azure service principal details. This is used for authenticating with Azure from GitHub Actions. (we created this in the previous step)

DOCKER_HUB_PASSWORD: Your Docker Hub password or access token if you're pushing images to Docker Hub.

repo_token: A GitHub token with necessary permissions for actions such as pushing container images to GitHub Container Registry.

Getting Started

!!! Make sure you have the necessary permissions and configurations set up in your Azure ML workspace and github repository. !!!

You will need to go into the pipelines folder and change what you need for your specific use case, adding your own image/data paths.

Same thing with the components folder, you will need to adjust the model and the preprocessing that I just went over to your specific use case.

3. Cloud AI Services

Azure Machine Learning Service was utilized extensively throughout the project. Azure ML provided a powerful platform for managing the entire MLOps pipeline, from data preparation to model registration. (It took a while to get used to working with azure ml, but once I got the hang of it, it was nice to see all the possibilities it offers)

3.1 Compute Resource Management

Compute resources were created and managed within Azure ML. This included setting up an Azure ML compute cluster instead of a compute instance to make the training faster (more nodes) since I had to preprocess 9 imagesets into a train/test split.

Compute Cluster

Cluster node status

- Idle
- Leaving
- Preparing
- Running

Cluster state

Allocation state: Succeeded (0 nodes)

Allocation state transition time: 12/11/2023, 6:57:55 PM

Created on: 12/11/2023, 1:43:12 PM

Current node count: 0

Attributes

Compute name: mllopsfinal

Resource ID: --

Compute type: Machine Learning compute

Subscription ID: 64410333-3fd9-440d-b9cd-d67e265e6a8c

Resource group: MLOps

Workspace: lucasmlops

Region: westeurope

Resource properties

Virtual machine size: Standard_E4s_v3 (4 cores, 32 GB RAM, 64 GB disk)

Processing unit: CPU - Memory optimized

Estimated cost: \$0.32/hr per node

OS Type: Linux

Virtual machine tier: Dedicated

Current available nodes: 3

Minimum number of nodes: 0

Maximum number of nodes: 3

Idle seconds before scale down: 120

Make sure to change the compute cluster to your own needs, this is what I used (found in ./environments/compute.yaml):

Computecluster:

```
ent > Y compute.yaml > ...
amlCompute.schema.json
$schema: https://azurermschemas.azureedge.net/latest/amlCompute.schema.json
name: mllopsfinal
type: amlcompute
size: STANDARD_E4S_V3
min_instances: 0
max_instances: 3
idle_time_before_scale_down: 120
tier: low_priority
location: westeurope
```

3.2 Environment Setup

Azure ML enabled the creation and management of environments for the project. Specifically, environment configurations for libraries like Pillow and TensorFlow that were defined in separate YAML files, found in the ./environments directory.

Environments

Create with this command:

```
az ml environment create --file my_env.yml --resource-group my-resource-group --workspace-name my-workspace
```

Where file is these files:

Pillow:

```

environment > Y pillow.yaml > YAML > {} tags
  environment.schema.json
  1 $schema: https://azuremlschemas.azureedge.net/latest/environment.schema.json
  2 description: Custom environment for Image Processing (with Pillow)
  3 name: aml-Pillow-cli
  4 version: 0.1.0
  5
  6 conda_file: ../components/dataprep/conda.yaml
  7 image: mcr.microsoft.com/azureml/openmpi4.1.0-ubuntu20.04:latest
  8 os_type: linux
  9
 10 tags:
 11   Pillow: 10.0.1
 12

```

Tensorflow:

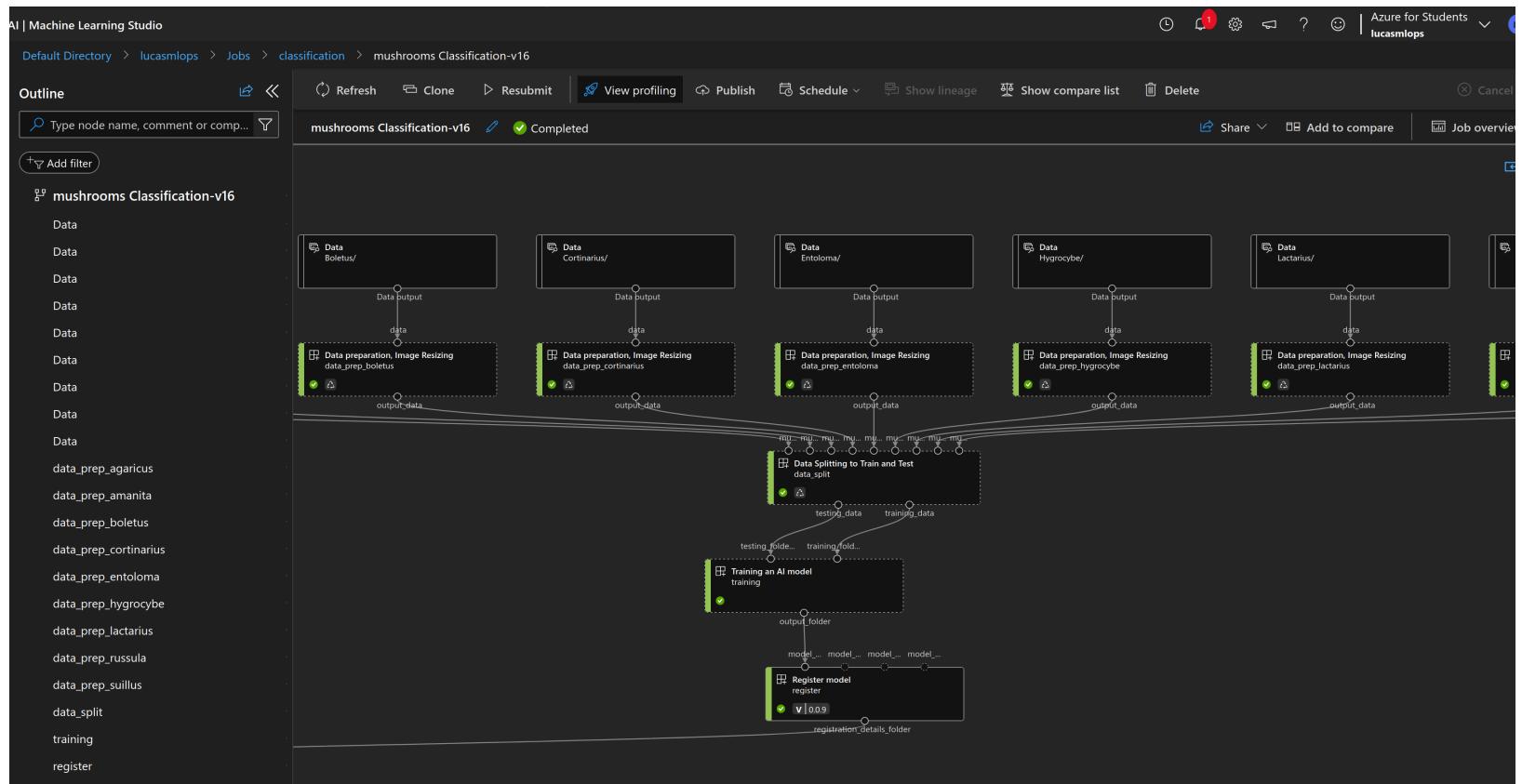
```

environment.schema.json
1 $schema: https://azuremlschemas.azureedge.net/latest/environment.schema.json
2 description: Custom Tensorflow Environment
3 name: aml-Tensorflow
4 version: 0.1.0
5
6 conda_file: ../components/training/conda.yaml
7 image: mcr.microsoft.com/azureml/openmpi4.1.0-ubuntu20.04:latest
8 os_type: linux
9
10 tags:
11   Pillow: 10.0.1
12

```

4. Model Training and Evaluation

4.1 Azure ML



4.2 Model Training

This is the 'main' component of training my model, here the model is build, compiled and trained.

```
## since the training time is already large the amount of epochs is kept really low (5
```

```
in this case)
INITIAL_LEARNING_RATE = 0.01
BATCH_SIZE = 32
PATIENCE = 11
model_name = 'mushroom-cnn'

opt = tf.keras.optimizers.legacy.SGD(lr=INITIAL_LEARNING_RATE,
decay=INITIAL_LEARNING_RATE / MAX_EPOCHS) # Define the Optimizer

model = buildModel((400, 400, 3), 9) # Create the AI model as defined in the utils
script.

#compile the model
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])

#data augmentation
aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
                         height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
                         horizontal_flip=True, fill_mode="nearest")

# train the network
history = model.fit( aug.flow(X_train, y_train, batch_size=BATCH_SIZE),
                      validation_data=(X_test, y_test),
                      steps_per_epoch=len(X_train) // BATCH_SIZE,
                      epochs=MAX_EPOCHS,
                      callbacks=[cb_save_best_model, cb_early_stop,
cb_reduce_lr_on_plateau] )
```

4.3 Model Evaluation

The performance of the trained model was evaluated using various metrics, including accuracy, precision, recall, and F1-score and a confusion matrix.

		precision	recall	f1-score	support					
	0	0.51	0.55	0.53	71					
	1	0.79	0.81	0.80	150					
	2	0.79	0.90	0.84	215					
	3	0.72	0.66	0.69	168					
	4	0.71	0.53	0.61	73					
	5	0.70	0.66	0.68	64					
	6	0.72	0.71	0.71	313					
	7	0.61	0.66	0.63	151					
	8	0.73	0.57	0.64	63					
	accuracy			0.71	1268					
	macro avg	0.70	0.67	0.68	1268					
	weighted avg	0.71	0.71	0.71	1268					
	[[39 7 4 1 2 0 12 5 1]]									
	[7 122 6 2 0 0 5 8 0]									
	[2 4 193 3 1 0 7 2 3]									
	[5 6 10 111 6 3 16 7 4]									
	[3 1 2 5 39 7 8 8 0]									
	[0 1 1 7 3 42 4 6 0]									
	[13 7 11 18 4 4 223 28 5]									
	[5 4 7 2 0 4 30 99 0]									
	[2 2 11 6 0 0 6 0 36]]									

4.3.1 conclusions

The model is performing pretty okay considering I only had around 259 images for 2 of the classes, my dataset wasn't balanced at all. (Adding data augmentation helped a lot with this)

And the training already took 5 hours, so it wasn't really feasible to add more data in my case, given the time frame of this project (and limited money resources).

5. Deployment

We were tasked with deploying a fastapi. I also added a webapp and a gradio gui to create a more visual experience.

5.1 Fastapi Deployment

The trained model is integrated into a fastapi, which is then built into a docker image and then finally that image is pushed to my github packages repo.

I later use this docker image to deploy the api on kubernetes.

Dockerfile for the fastapi

```
inference > 📁 dockerfile > ...
1 FROM python:3.10
2
3 WORKDIR /app
4
5 COPY requirements.txt .
6 RUN pip install -r requirements.txt
7
8 COPY ./ ./
```

10 # CMD ["python", "main.py"]

11 CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]

- **Deployment and service file for the fastapi**

```
io.k8s.api.apps.v1.Deployment (v1@deployment.json)
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: api-deployment
5 spec:
6   replicas: 1
7   selector:
8     matchLabels:
9       app: api
10  template:
11    metadata:
12      labels:
13        app: api
14  spec:
15    containers:
16      - name: api
17        image: ghcr.io/driessenslucas/mlops-mushrooms-api:latest
18        ports:
19          - containerPort: 8000
20        resources:
21          requests:
22            memory: '1Gi'
23            cpu: '500m'
24          limits:
25            memory: '4Gi'
26            cpu: '2'
```

```

1 io.k8s.api.core.v1.Service (v1@service.json)
2 # service.yaml
3 apiVersion: v1
4 kind: Service
5 metadata:
6   name: api-service
7 spec:
8   selector:
9     app: api
10  ports:
11    - protocol: TCP
12      port: 80
13      targetPort: 8000
14    type: NodePort

```

5.3 API Endpoints

The screenshot shows a browser window with the URL `localhost:8700/docs`. The page title is "FastAPI 0.1.0 OAS 3.1". Below the title, there is a link to `/openapi.json`.

The main content area is titled "default". It lists three API endpoints:

- `GET /gradio_exists` Gradio Exists
- `POST /upload/image` Uploadimage
- `GET /healthcheck` Healthcheck

Below the endpoints, there is a section titled "Schemas" which contains definitions for `Body_uploadImage_upload_image_post`, `HTTPValidationError`, and `ValidationError`.

5.4 Web App

This simple web app creates a fun and interactive way to test the finished product, its made to do an api request with an uploaded image, based on the result it will display some information about the mushroom. (I added text to speech to spice it up a bit)

Look at the code snippet below to see how the api call is done.

```

classifyButton.addEventListener('click', function () {
  //get image data from canvas
  const imageDataURL = document.getElementById('mushroomImage').src;
  //send image data to api
  fetch(imageDataURL)
    .then((res) => res.blob())
    .then((blob) => {
      // Create a FormData object
      const formData = new FormData();

```

```

        formData.append('img', blob, 'image.png');

        // Send the image file to the FastAPI server
        fetch('http://api-service/upload/image', {
            method: 'POST',
            body: formData,
        })
        .then((response) => response.json())
        .then((mushroomFamily) => {
            console.log(mushroomFamily);

            // Update the UI with the received data
            mushroomName.textContent = mushroomFamily || 'Unknown';
            mushroomType.textContent = 'Mushroom Family';
            mushroomDescription.textContent =
                mushroomDescriptions.find((x) => x.name === mushroomFamily)
                .description || 'No description available';

            //make sound button available
            soundButton.style.display = 'block';
        })
        .catch((error) => {
            console.error('Error:', error);
        });
    });
}

```

Dockerfile for the web app

```

web > dockerfile > ...
1 # Use the Nginx image from Docker Hub
2 FROM nginx:alpine
3
4 # Copy the static website files into the default Nginx public folder
5 COPY ./voice-overs/ /usr/share/nginx/html/voice-overs/
6 COPY index.html /usr/share/nginx/html/
7 COPY styles.css /usr/share/nginx/html/
8 COPY script.js /usr/share/nginx/html/
9
10 # Expose port 80
11 EXPOSE 80
12
13 # Start Nginx when the container has provisioned
14 CMD ["nginx", "-g", "daemon off;"]
15

```

Deployment and service for the web app on kubernetes

```

1  io.k8s.api.apps.v1.Deployment (v1@deployment.json)
2  apiVersion: apps/v1
3  kind: Deployment
4  metadata:
5    name: website-deployment
6  spec:
7    replicas: 1
8    selector:
9      matchLabels:
10     app: website
11   template:
12     metadata:
13       labels:
14         app: website
15   spec:
16     containers:
17       - name: website
18         image: ghcr.io/driessenslucas/mlops-mushrooms-website:latest
19         imagePullPolicy: Always
20         ports:
21           - containerPort: 80
22         resources:
23           requests:
24             memory: '2Gi'
25             cpu: '500m'
26           limits:
27             memory: '4Gi'
28             cpu: '2'
29
30
31  io.k8s.core.v1.Service (v1@service.json)
32  apiVersion: v1
33  kind: Service
34  metadata:
35    name: web-service
36  spec:
37    selector:
38      app: website
39    ports:
40      - protocol: TCP
41        port: 80
42        targetPort: 80
43    type: NodePort

```

5.6 Gradio

I also used Gradio to build a simpler gui, since the webapp was made more for fun and as a potential software integration.

This was implemented in the fastapi app (which wasn't easy to do.... Once the api starts, there will be an endpoint at /gradio, which will then show the gradio interface.)

```

async def gradio():
    # implement gradio
    with gr.Blocks() as demo:

        # Function to make predictions using the loaded model
        def predict(image):
            original_image = image
            original_image = original_image.resize((400, 400))
            images_to_predict = np.expand_dims(np.array(original_image), axis=0)
            predictions = model.predict(images_to_predict)

```

```

#get the highest probability
classifications = predictions.argmax(axis=1)

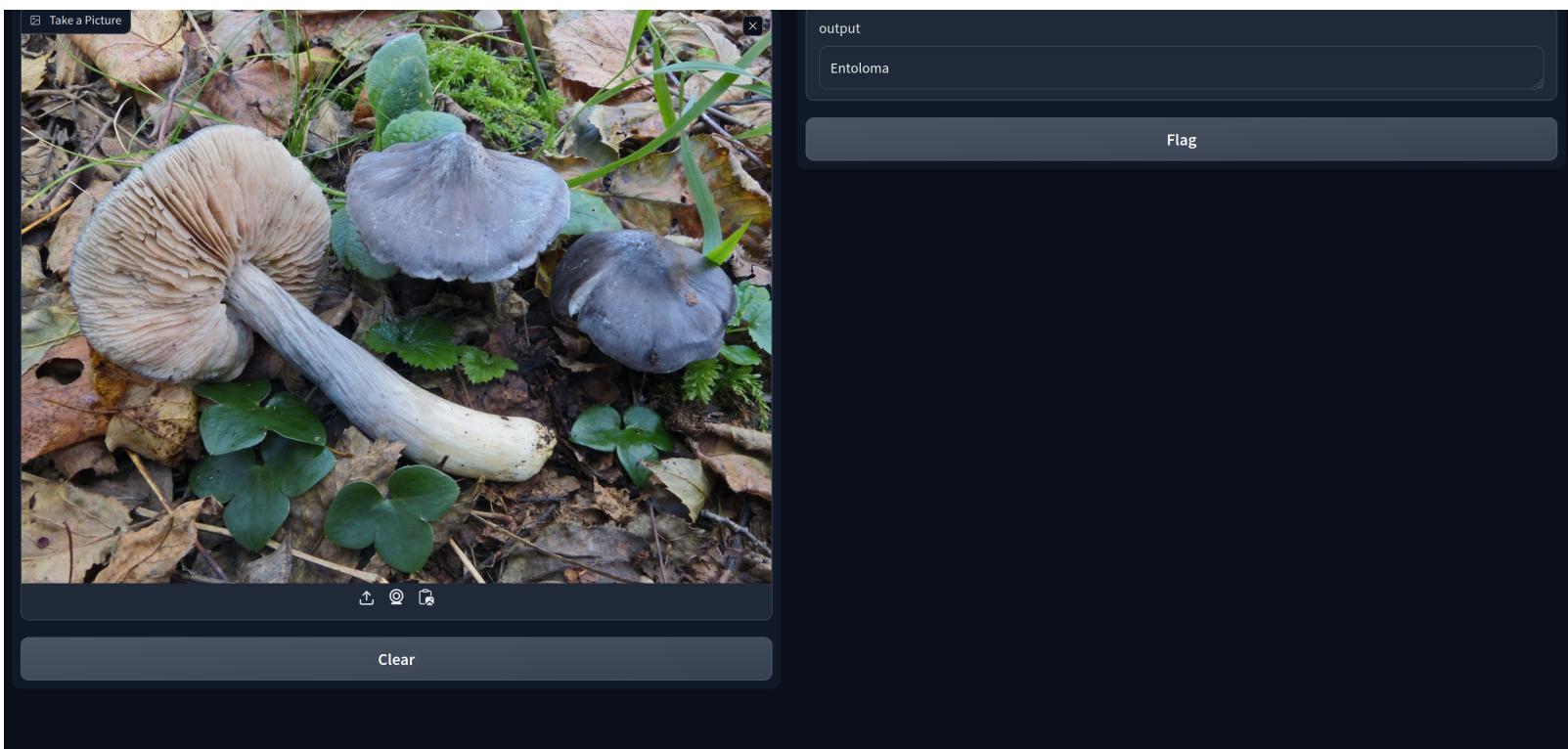
# Print probability of all classes
print(predictions)

#get the name of the mushroom after the prediction
return f'{Mushrooms[classifications.tolist()[0]]}'

# Gradio Interface for Mushroom Prediction
iface = gr.Interface(
    fn=predict,
    inputs=gr.Image(type='pil', label='Take a Picture'),
    outputs='text',
    live=True
)

# Run Gradio Interface in the background
global app
demo.queue()
demo.startup_events()
app = gr.mount_gradio_app(app, demo, '/gradio')

```



5.7 Deployment on azure kubernetes !!! (bonus)

This step was in my eyes the whole point of this assignment, creating a full pipeline from data to webapp, that is fully automated and deployed on a kubernetes cluster

Kubernetes Cluster Setup

note:this has been barely tested since I'm limited in public ip's that azure will give me

As I wasn't sure if this was part of the assignment, I added this as a bonus, I deployed the api and the website on azure kubernetes (instead of on my own machine), this was done by adjusting the github actions file.

For this I changed my strategy a bit, I changed from working with a clusterIP + port forwarding to a loadbalancer, this way I could set up an external ip in azure and access the api and website from anywhere (I should probably added some type of auth, but since its only able to upload a picture, it should be fine).

I had to add some new env variables in the github actions file, for this to work (this is the full updated env section):

```

env:
  NAMESPACE: mushroomspace
  GROUP: MLOps
  CLUSTER: mushrooms
  WORKSPACE: lucasmlops
  LOCATION: westeurope
# Allow to override for each run, in the workflow dispatch manual starts
CREATE_COMPUTE: ${{ github.event.inputs.create_compute }}
TRAIN_MODEL: ${{ github.event.inputs.train_model }}
SKIP_TRAINING_PIPELINE: ${{ github.event.inputs.skip_training_pipeline }}
DEPLOY_MODEL: ${{ github.event.inputs.deploy_model }}
DOWNLOAD_MODEL: ${{ github.event.inputs.download_model }}
DEPLOY_KUBERNETES: ${{ github.event.inputs.deploy_kubernetes }}
CREATE_CLUSTER: ${{ github.event.inputs.create_cluster }}

```

- I then added a new step in the azure-pipeline job:

```

- name: Create Kubernetes cluster
uses: azure/CLI@v1
id: prepare-kubernetes-cluster
if: ${{ inputs.create_cluster }}
with:
  azcliversion: 2.53.0
  inlineScript: |
    az aks create -g $GROUP -n $CLUSTER --enable-managed-identity --node-count 1
--enable-addons --enable-msi-auth-for-monitoring --generate-ssh-keys

```

- And also added a new job responsible for deploying the api and the website on the kubernetes cluster:

```

deploy-kubernetes:
  needs: deploy
  if: ${{ inputs.deploy_kubernetes }}
  runs-on: ubuntu-latest
  steps:
    - name: Check out repository
      uses: actions/checkout@v4

    - name: Azure Login
      uses: azure/login@v1
      with:
        creds: ${{ secrets.AZURE_CREDENTIALS }}

    - name: Set AKS context
      uses: azure/aks-set-context@v1
      with:
        creds: ${{ secrets.AZURE_CREDENTIALS }}
        cluster-name: ${{ env.CLUSTER }}
        resource-group: ${{ env.GROUP }}

    - name: Create Namespace or check namespace
      run: |
        kubectl create namespace $NAMESPACE || echo "namespace already exists"

    - name: deploy website and fastapi onto the kubernetes
      run: |
        kubectl apply -f ./web/deployment.yaml -n $NAMESPACE
        kubectl apply -f ./inference/deployment.yaml -n $NAMESPACE

```

- To ensure that the api and website are kept up-to-date I added a 'rolling-update' strategy to the deploy step (where the images get reuploaded to the github packages repo)
- The deployment/*-deployment name is the metadata.name in the deployment.yaml files.

```
- name: Update Kubernetes Deployment
  run: |
    kubectl set image deployment/api-deployment api=ghcr.io/driessenslucas/mllops-
mushrooms-api:latest -n $NAMESPACE
    kubectl set image deployment/website-deployment website=ghcr.io/driessenslucas/mllops-
mushrooms-website:latest -n $NAMESPACE
```

- Proof of this working (look at the ip's):

Microsoft Azure

mushrooms | Services and ingresses

Name	Namespace	Status	Type	Cluster IP	External IP	Ports	Age
kubernetes	default	Ok	ClusterIP	10.0.0.1	443/TCP	13 minutes	...
kube-dns	kube-system	Ok	ClusterIP	10.0.0.10	53/UDP, 53/TCP	12 minutes	...
metrics-server	kube-system	Ok	ClusterIP	10.0.88.212	443/TCP	12 minutes	...
calico-typha	calico-system	Ok	ClusterIP	10.0.46.91	5473/TCP	10 minutes	...
web-service	default	Ok	LoadBalancer	10.0.97.218	4.175.227.159, 80.31576/TCP	10 minutes	...
api-service	default	Ok	LoadBalancer	10.0.81.158	4.175.227.165, 80.32097/TCP	10 minutes	...

MUSHROOMDEX

Take or upload a picture of a mushroom to classify it.

Name: Unknown

Type: Unknown

Description: Unknown

Hear Mushroom Info:

The screenshot shows the FastAPI documentation page at 4.175.227.165/docs. The title is "FastAPI 0.1.0 OAS 3.1". Below the title, there is a link to [/openapi.json](#). The main content area is titled "default". It lists three API endpoints:

- GET /gradio_exists** Gradio Exists
- POST /upload/image** Uploadimage (highlighted in green)
- GET /healthcheck** Healthcheck

Below the endpoints is a section titled "Schemas" which contains three schema definitions:

- `Body_uploadImage_upload_image_post` > Expand all `object`
- `HTTPValidationError` > Expand all `object`
- `ValidationError` > Expand all `object`

6. Integration with Existing Software

In a practical scenario, this MLOps pipeline is ready to be integrated into an existing software system.

MUSHROOMDEX



Name:

Agaricus

Type:

Mushroom Family

Description:

This genus is known for mushrooms with a fleshy cap and a number of radiating gills underneath, where spores are produced. They are characterized by chocolate-brown spores and a stem or stipe that elevates the mushroom above its growing substrate. Agaricus includes

Hear Mushroom Info:



Upload Image

Classify

6.1 Fake company

As I didn't really have an actual fake company in mind, I created a webapp (as shown before) but this could be used in a lot of different ways, for example, a company that wants to classify mushrooms for their restaurant, or a company that wants to classify mushrooms for their mushroom farm, or a company that wants to classify mushrooms for their mushroom picking tours.

If there was a better more in depth dataset this could even be used to classify mushrooms in the wild, and help people identify mushrooms (to see if they are edible or not).

6.1.1 Integration in an existing software system

When you want to integrate this you just need to do an api call to the fastapi endpoints.... So there isn't much to it, you could deploy the api to a container app in azure for example. Or in a cloud kube cluster, like I did as a bonus.

7. Automation and Version Control

In this project I mainly used GitHub Actions which allows for the orchestration of various tasks, from data handling to model training and deployment, all triggered by specific GitHub events like code commits.

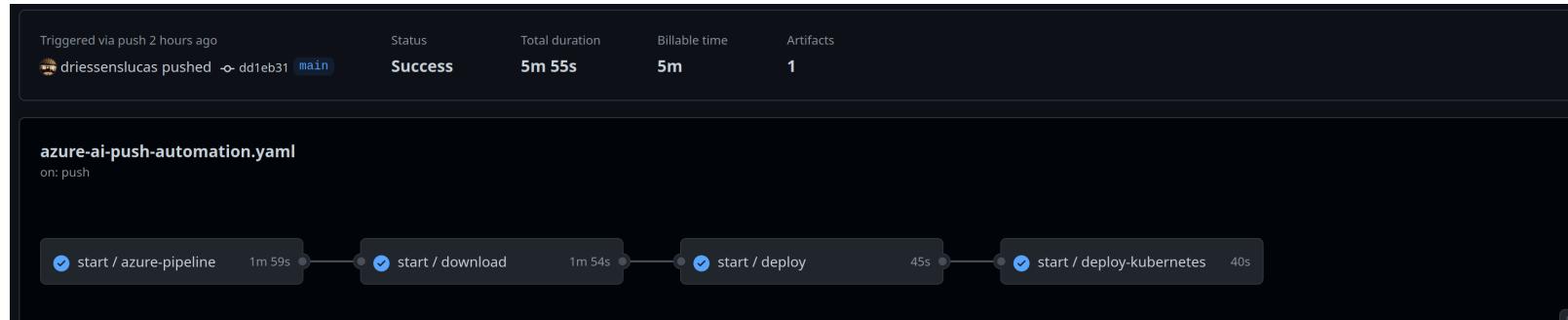
Automation and version control are found all over my MLOps pipeline. The key components of my automation strategy include:

Data Handling: Automating the extraction and preprocessing of data to prepare it for training.
Model Training: Triggering the training process of the machine learning model with specified parameters.
Model Evaluation: Systematically evaluating the model performance to ensure it meets our criteria.
Model Deployment: Deploying the model to my GitHub Packages repository for easy access.
Version Control: Using GitHub Actions to automate the version control of the model.

In the following sections, I delve into the specifics of each step, illustrating how GitHub Actions enhances our MLOps pipeline's efficiency and robustness.

7.1 GitHub Actions

I used GitHub actions to automate the training and deployment of the model. The workflow is defined in the .github/workflows directory. It triggers a pipeline that goes the whole process of data extraction, preprocessing, training, evaluation, and deployment. each directory has its own yaml file for this.



Pipeline start

Here you can set the environment variables for the pipeline, choosing if you want to create_compute, train_model, skip_training_pipeline, download_model or deploy_model these allow for a more flexible pipeline, where you can choose to skip certain steps. since you don't need to recreate the compute or train the model each time you want to redeploy it.

```

name: Azure ML Automated Pushing pipeline
on:
  push:
    branches:
      - main
jobs:
  start:
    uses: ./github/workflows/azure-ai.yaml@v1
    with:
      # Set these to "true" if you want to run these pipeline steps or "false" to skip it
      create_compute: false
      train_model: false
      skip_training_pipeline: true
      deploy_model: false
      download_model: false
      deploy_kubernetes: true
      create_cluster: false
    secrets: inherit

```

Jobs

azure cli

This job will login and create and/or start the compute cluster. and start the training pipeline if selected (./pipelines/mushroom-classification.yaml).

```

jobs:
  azure-pipeline:
    if: ${{ inputs.skip_training_pipeline }}
    runs-on: ubuntu-latest
    # runs-on: self-hosted
    outputs:
      ai-model-version: ${{ steps.azure-ml-pipeline.outputs.latest_version }}
    steps:
      - name: Check out repository...
      - name: Azure Login...
        # AZURE_CREDENTIALS should be a secret in your repo containing a JSON string of your service principal
      - name: Install YQ...
      - name: read-yaml-file...
      - name: Create compute...
      - name: Create Kubernetes cluster...
      - name: Start compute...
      - name: Execute Azure ML Script...
      - name: Cleanup Compute...

```

download model

This will download the registered model from azure ml and save it in the ./inference directory on the created machine.

```

download:
  needs: azure-pipeline
  # Only run if azure-pipeline is succeeded OR skipped
  if: ${{ needs.azure-pipeline.result == 'success' || needs.azure-pipeline.result == 'skipped' }} && i
  # runs-on: self-hosted
  runs-on: ubuntu-latest
  steps:
    - name: Check out repository...
    - name: Azure Login...
    - name: Set model version...
    - name: Upload API Code for Docker...

```

deploy model

This will deploy the docker files to the github packages repo

```

deploy:
  needs: download
  # Only run if download is succeeded OR skipped AND if the deploy_model variable is true
  if: ${{ needs.download.result == 'success' || needs.download.result == 'skipped' }} && inputs.deploy_model
  runs-on: self-hosted
  # runs-on: ubuntu-latest
  steps:
    - name: Gather Docker Meta Information...
      # Enter your GITHUB Token here!
    - name: Login to GHCR...
    - name: Download API Code for Docker...
    - name: Docker Build and push...
    - name: Docker build and push website image...
    - name: Update Kubernetes Deployment...

```

All this can then be tested (locally) with the test.yaml file

Testing file

(This will require a local github actions runner)

This will deploy the api and the website on kubernetes. It will create a new namespace and then port forward both the api and the website to the localhost, allowing the user to explore. After testing is done it removes the namespace and all the containing services.

```

name: Test Local Runner

on:
  workflow_dispatch:
  push:
    branches:
      - main

jobs:
  default-job-test:
    runs-on: self-hosted

    steps:
      - name: Check out repository ...
      - name: Set up Kubectl ...
      - name: Clean up previous namespaces ...
      - name: Create Test Namespace ...
      - name: Deploy to Kubernetes for Testing ...
      - name: Deploy Website to Kubernetes for Testing ...
      - name: Get Deployment Pods and Wait for Them to Be Ready ...
      - name: Execute Tests ...
      - name: Cleanup ...

```

7.2 Version Control

Version control is mandatory for any project, but especially when wanting to create a pipeline that can be used in a production environment. without it, you would 100% run into problems.

Here I pasted some snippets of the version controlling I used in this project.

7.2.1 Training model version

When training the model, the name is set using `github.sha` and `github.run_id`, this ensures that each model has a unique name, and that the model can be traced back to the commit and run that created it.

```

- name: Execute Azure ML Script
uses: azure/CLI@v1
id: azure-ml-pipeline
if: ${{ inputs.train_model }}
with:
azcliversion: 2.53.0
inlineScript: |
  az extension add --name ml -y
  az configure --defaults group=$GROUP workspace=$WORKSPACE location=$LOCATION
  az ml job create --file ./pipelines/mushroom-classification.yaml --set name=mushrooms-
classification-${{ github.sha }}-${{ github.run_id }} --stream

```

7.2.2 Model Version Retrieval

Within the "download" step of the GitHub Actions workflow, model version retrieval is performed. This step ensures that the latest version of the trained AI model is obtained from the Azure Machine Learning workspace:

```

- name: Set model version
uses: azure/CLI@v1
with:

```

```

azcliversion: 2.53.0
inlineScript: |
    az extension add --name ml -y
    az configure --defaults group=$GROUP workspace=$WORKSPACE location=$LOCATION
    VERSION=$(az ml model list --name mushroom-classification --resource-group $GROUP
    --workspace-name $WORKSPACE --query "[0].version" -o tsv)
    az ml model download --name mushroom-classification --download-path ./inference
    --version $VERSION --resource-group $GROUP --workspace-name $WORKSPACE

```

In this step, the `az ml model list` command retrieves the version information for the "mushroom-classification" model. This version is then used to download the corresponding model artifacts to the specified path.

7.2.3 Model Artifact Storage

The downloaded model artifacts are stored within the "inference" directory, making them easily accessible for deployment and inference.

```

- name: Download API Code for Docker
  uses: actions/download-artifact@v2
  with:
    name: docker-config
    path: inference

```

The "docker-config" artifact, which includes the downloaded model, is made available for subsequent steps, such as Docker containerization and deployment.

7.2.4 Model Deployment Version

When deploying the model, the name is set using the `:latest` tag (a v1.0 type of tag would be better, but this works).

This makes it easy to always ensure you have the latest version of the model.

```

- name: Docker Build and push
  id: docker_build
  uses: docker/build-push-action@v2
  with:
    context: ./inference
    push: true
    tags: ghcr.io/driessenslucas/mlops-mushrooms-api:latest

```

8. Conclusion

In conclusion, this MLOps project effectively demonstrated the principles of developing, training, deploying, and testing a machine learning model for mushroom classification. Leveraging Azure ML, FastAPI, and GitHub Actions, the project showcased an end-to-end pipeline that can be easily adapted to real-world scenarios.

I had a lot of fun learning while doing this project, I hope my documentation is adequate at explaining what I did and why I did it 😊

9. Useful links

- How to get github token: <https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens>
- How to use github secrets <https://docs.github.com/en/actions/security-guides/using-secrets-in-github-actions>
- How to create azure service principle to access azure services: <https://learn.microsoft.com/en-us/cli/azure/azure-cli-service-principal-tutorial-1?tabs=bash>
- Kubectl cheat sheet (for debugging): <https://www.bluematador.com/learn/kubectl-cheatsheet>

- create a Kubernetes cluster in azure azk: <https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-cli>
- How to setup a github actions runner:<https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/adding-self-hosted-runners>

Demos

(Use sound for the web app demo to hear the description being read out loud)

<https://github.com/driessenslucas/MLOps-pipelines-2023-main/assets/91117911/654ceb40-4fa5-4354-b318-4921450ee955>

<https://github.com/driessenslucas/MLOps-pipelines-2023-main/assets/91117911/c75f7b37-0d1c-4acc-b045-bb258e13633c>

<https://github.com/driessenslucas/MLOps-pipelines-2023-main/assets/91117911/cd7ff6f4-3cc3-468a-aed1-70c9643b79da>

Download model (optional)

(if your model file exceeds 100 MB, You should be using the AzCopy tool for this instead.)

```
az ml model download --name ${name} --version ${version} --download-path ${path}  
--resource-group ${group} --workspace-name ${workspace}
```