Euro Currency Note Identification Using AWS Sage Maker

Cloud Computing SS2022

Submitted by:

Moeez Ur Rehman
Muhammad Hasseb Anwar
Sharish Kanwal
Harmain Haider

Under the guidance of:

Prof. Dr. Christian Baun

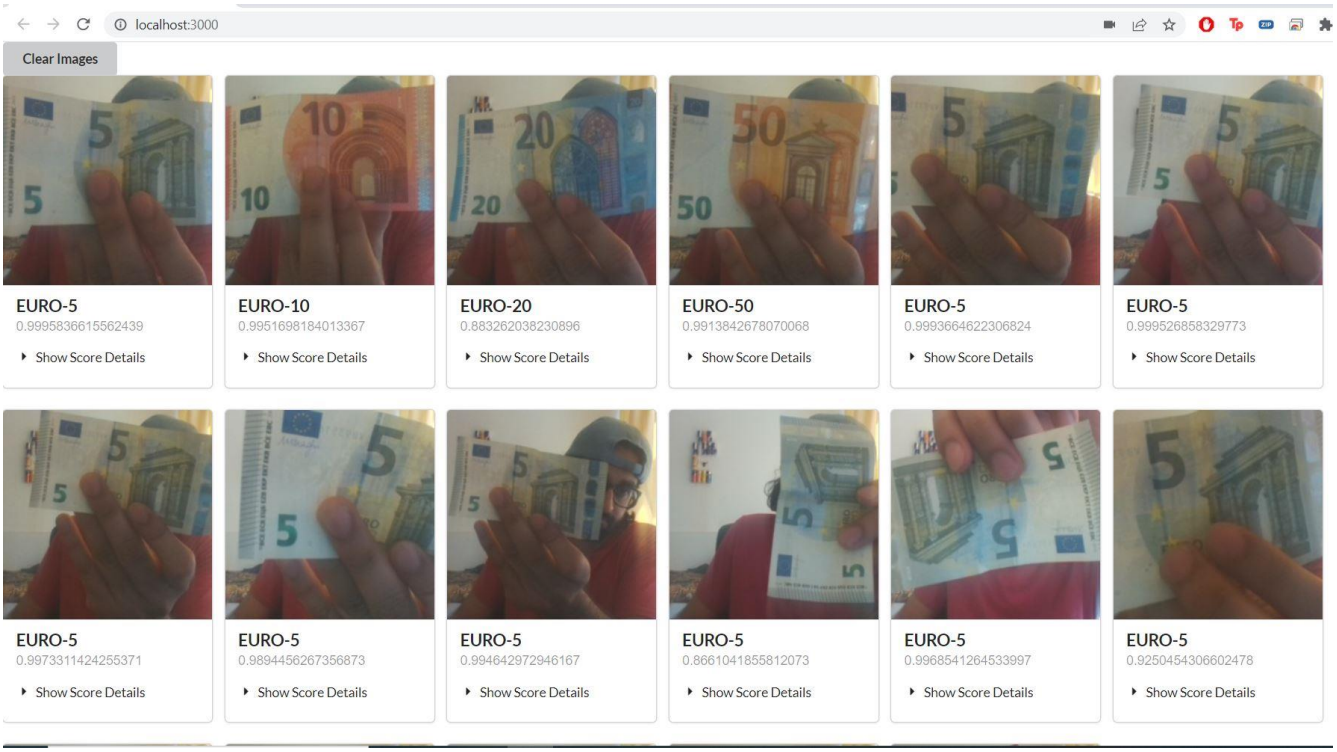
Table of Content

1. Introduction

The goal of the project is to implement machine learning model on the cloud service, we have used AWS cloud platform for this project, The problem statement was to identify different currency notes, our training data is stored on AWS S3 from where is being pulled by AWS's Sagemaker service to train our model.

After the training the endpoint is being deployed on sagemaker which then be used by our webapp to predict out currency notes. For the sake of simplicity, we have trained our model for only Euro 5,10,20 and 50 notes.

To train our model we have used **Sagemaker's built in Image classification Algorithm** which is based on **Supervised Learning** that supports multi-class classification. It uses Conventional Neural Networks (CNN). More info on the image classification model of sagemaker can be access using this link https://docs.AWS.amazon.com/sagemaker/latest/dg/image-classification.html



2. Technologies

2.1 AWS SageMaker

It is fully managed machine learning services. Machine learning models easily train, build and deployed directly into a production ready hosted environment with Sage Maker. It provides Jupyter notebook instance for easy access to your data sources for exploration and analysis, so you don't have to manage servers. It also provides common machine learning algorithms that we can run efficiently against extremely large data in a distributed environment. With native support we can also bring-our-own-algorithms and frameworks, SageMaker offers flexible distributed training options that adjust to your specific systems. Deploy a model into a secure and scalable environment by launching it with a few clicks from SageMaker Studio or the SageMaker console. Training and hosting are billed by minutes of usage, with no minimum fees and no upfront commitments.

2.2 SageMaker End point

An Amazon SageMaker endpoint is a fully managed service that allows you to make real-time inferences via a REST API. Taking the pain away from running your own EC2 instances, loading artefacts from S3, wrapping the model in some lightweight REST application, attaching GPUs and much more. This is great as it means with a single click or command you have a fully working solution.

2.3 AWS Lambda functions

When we write our code, we are responsible for our code only. Lambda manages network, memory, CPU and other resources to run code. Lambda manages resources because we cannot log in to the compute instances or customize the operating system on provided runtimes. On our behave lambda perform administrative and operational activities such as monitoring, managing capacity etc. By using Lambda API we invoke our lambda function.

We can use Lambda to:
Create our own backend that operates at AWS scales, performance and security.
Build data processing triggers for AWS services such as Amazon Simple Storage Services (Amazon S3)

2.4 AWS API Gateway

It is a service for creating, maintaining, publishing and securing REST, HTTP, and WebSocket APIs at any scale. API developers can create APIs that access AWS or other web services, as well as data stored in the AWS Cloud. As an API Gateway API developer, you can create APIs for use in your own client applications.

API endpoint

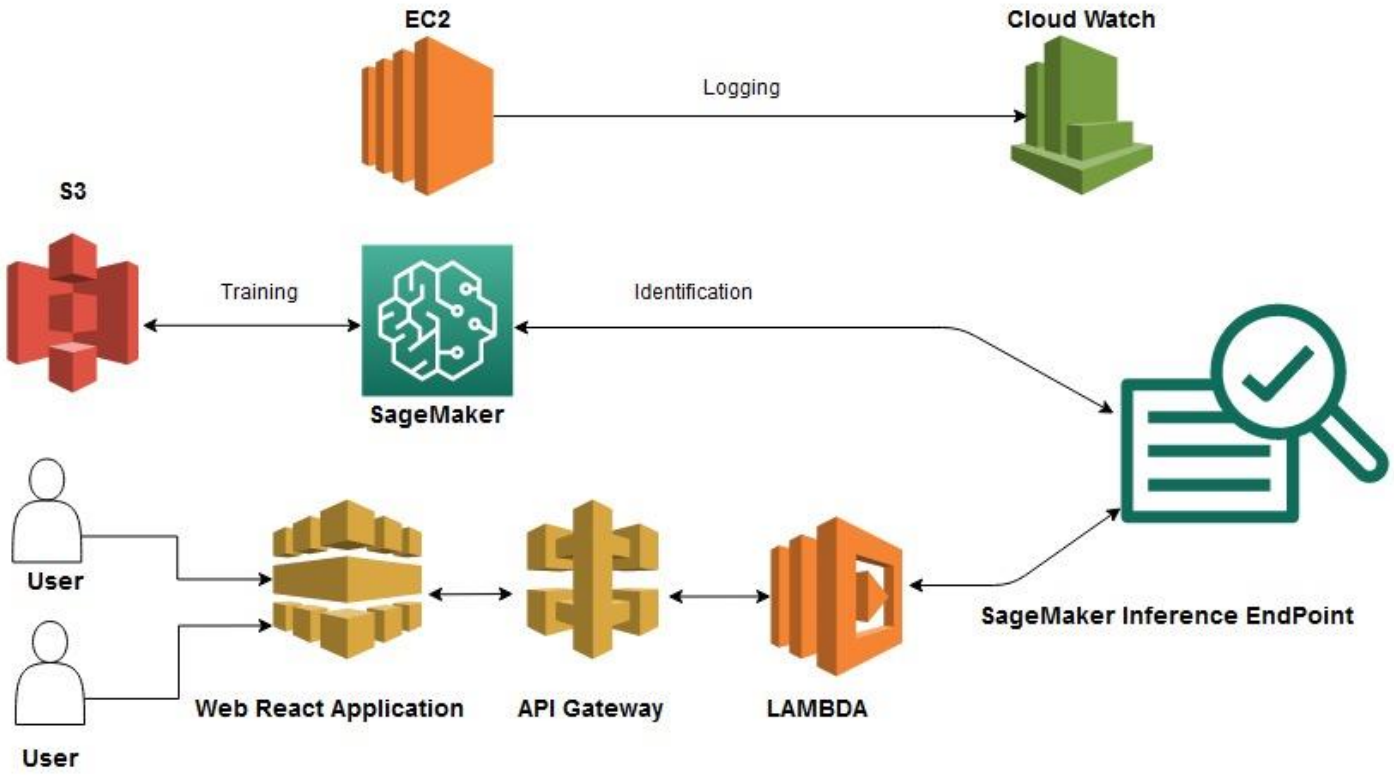
API endpoints are the specific digital location to retrieve the digital resource that exists there when request for information are sent by one program. To guarantee the proper functioning of the incorporated software, Endpoint specify where APIs can access resources.

2.5 AWS Amplify

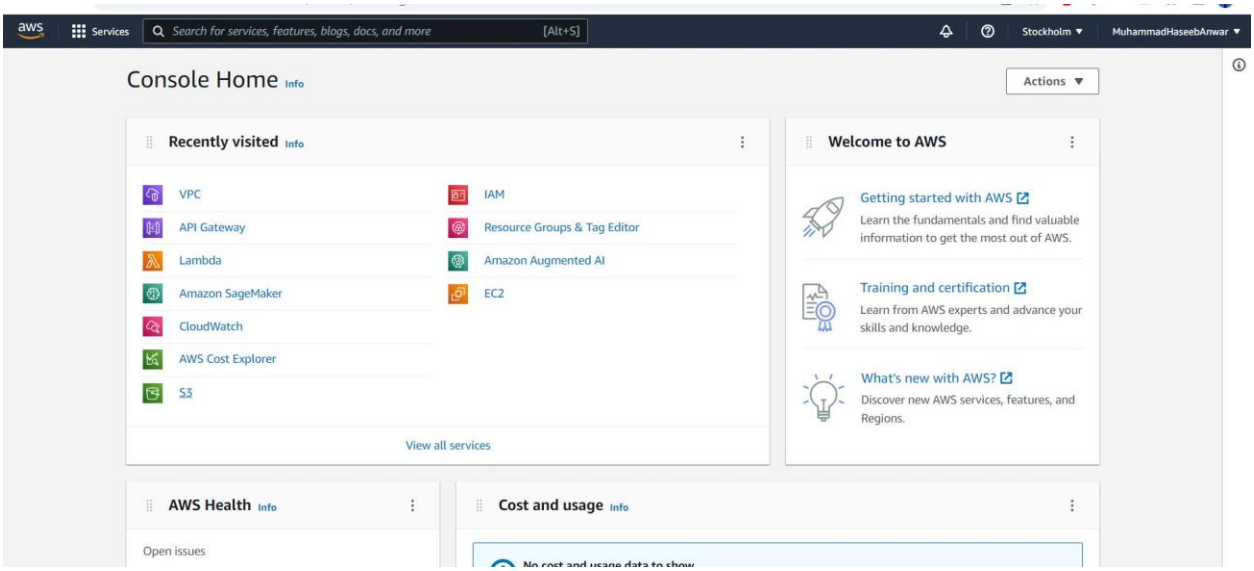
AWS Amplify include ready to use components, code lines and built-in command line interface designed to help developers easily create and launch apps. It also allows you to securely and quickly integrate a wild range of functions ranging from API to AI. It is a full stack application platform with both client side and server-side code.

Various AWS services used in our project, to train the model we have used S3 to store our data and Sagemaker to train our model

2.6 Flow Diagram

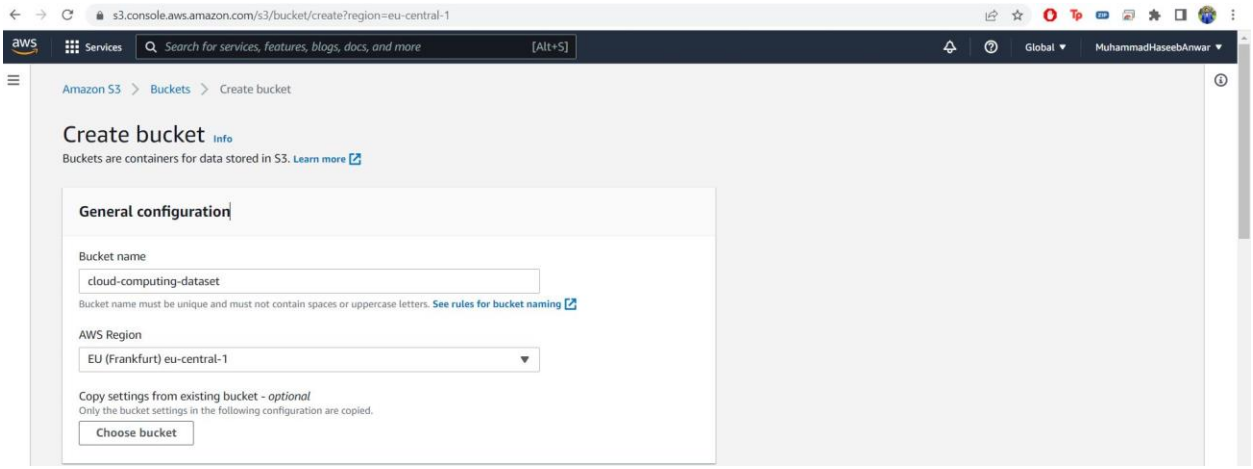


3. Training model on Sagemake

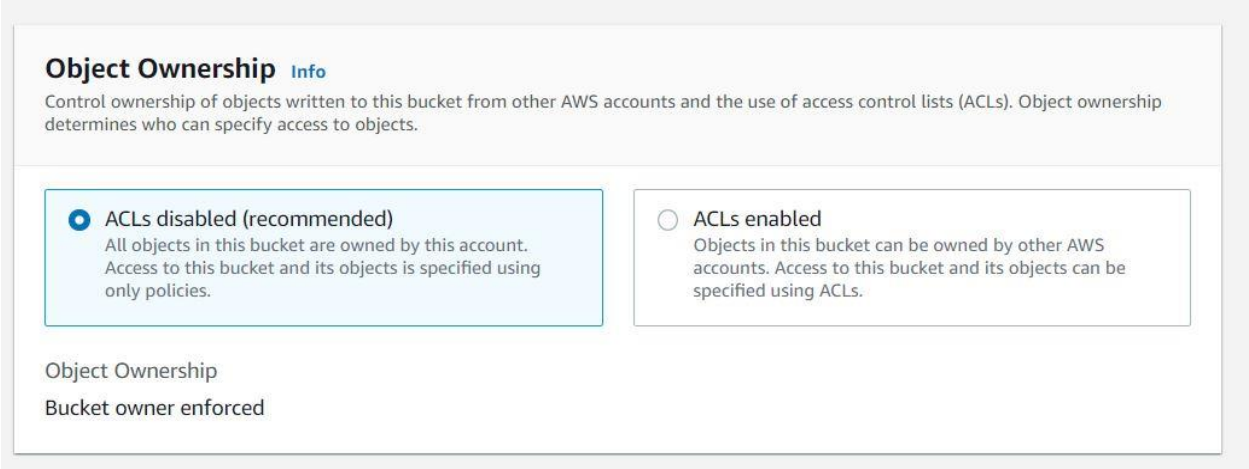


3.1.1 Creating S3 bucket

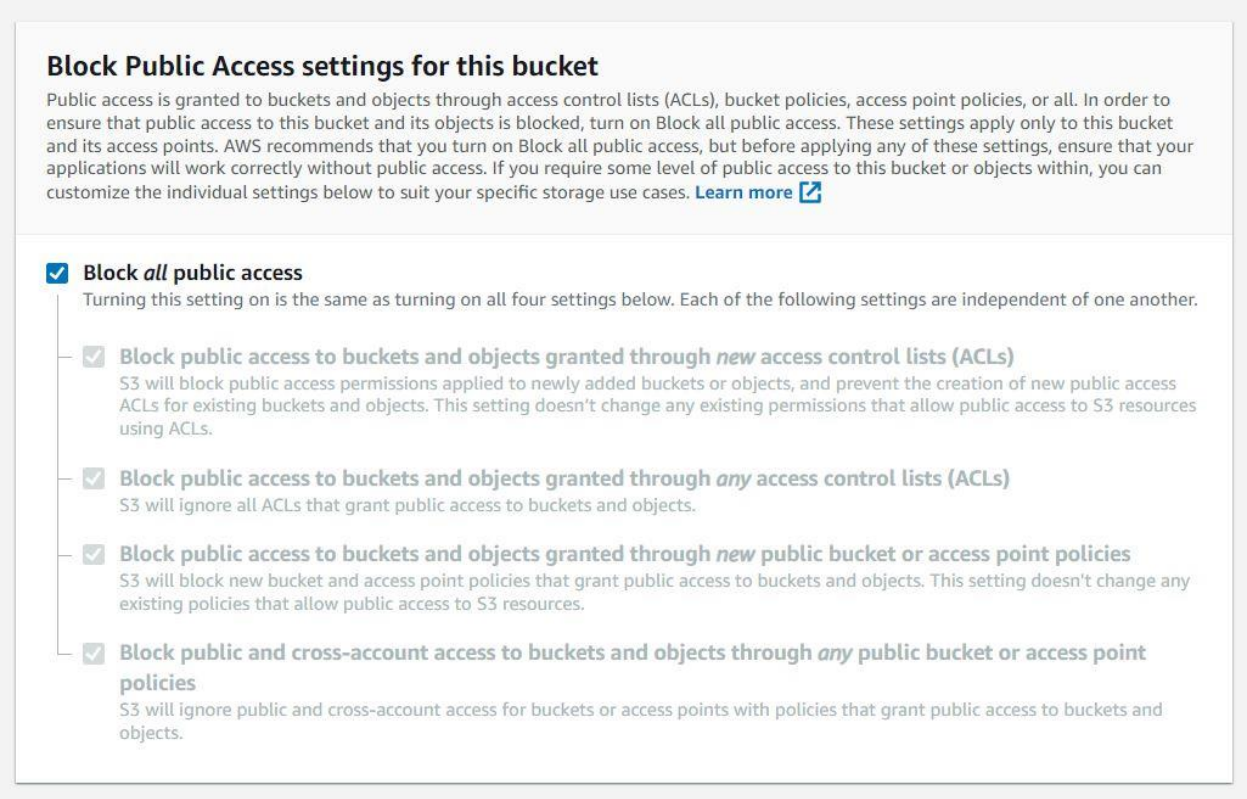
The very first task is to create a S3 bucket, we have to provide a name of the bucket, choose the region, in our case we have chosen eu-central-1 location which is in Frankfurt.



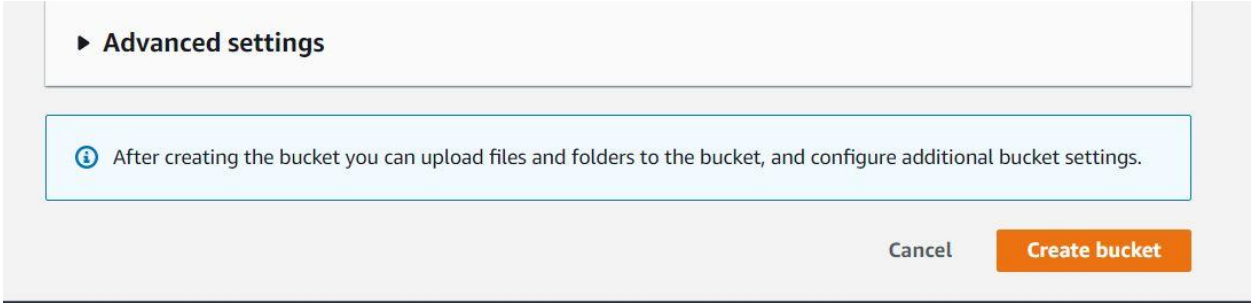
Here we need to implement the object ownership, if we want the object of this bucket to be owned by multiple accounts then we should choose ACLs Enabled, Although the recommended property is **ACLs disabled.**



Here we can implement the security of our bucket, it is always recommended to block all public access of the bucket.

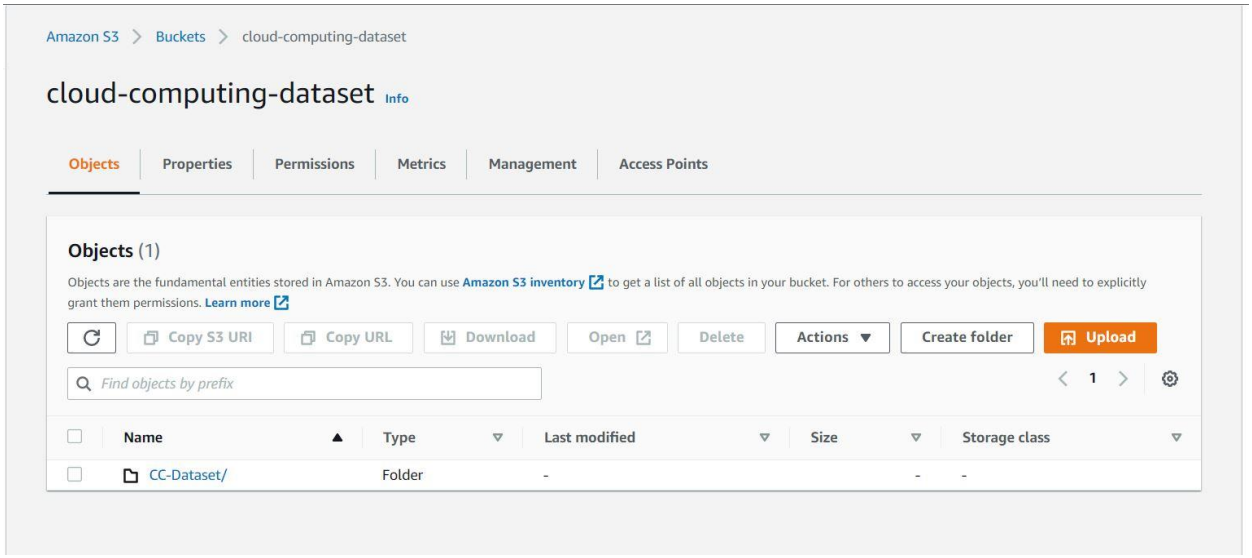


Just click on create bucket and our bucket will be created.



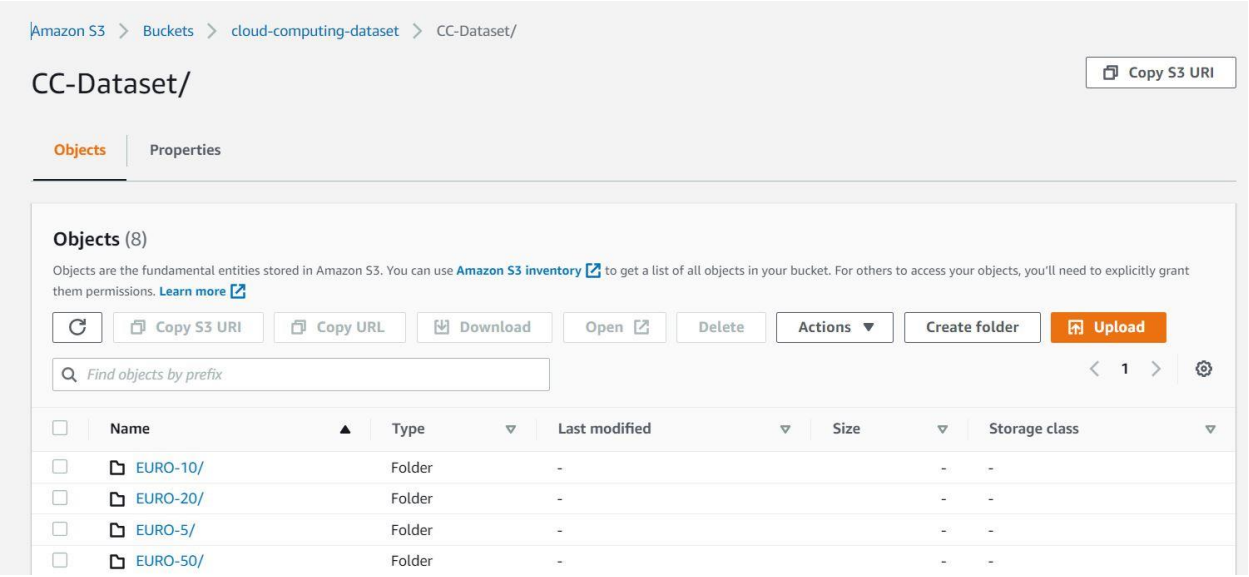
3.1.2 Uploading training images on s3

Next step is to create folders within the S3 bucket. We have created our main folder with the name 'CC_Dataset'

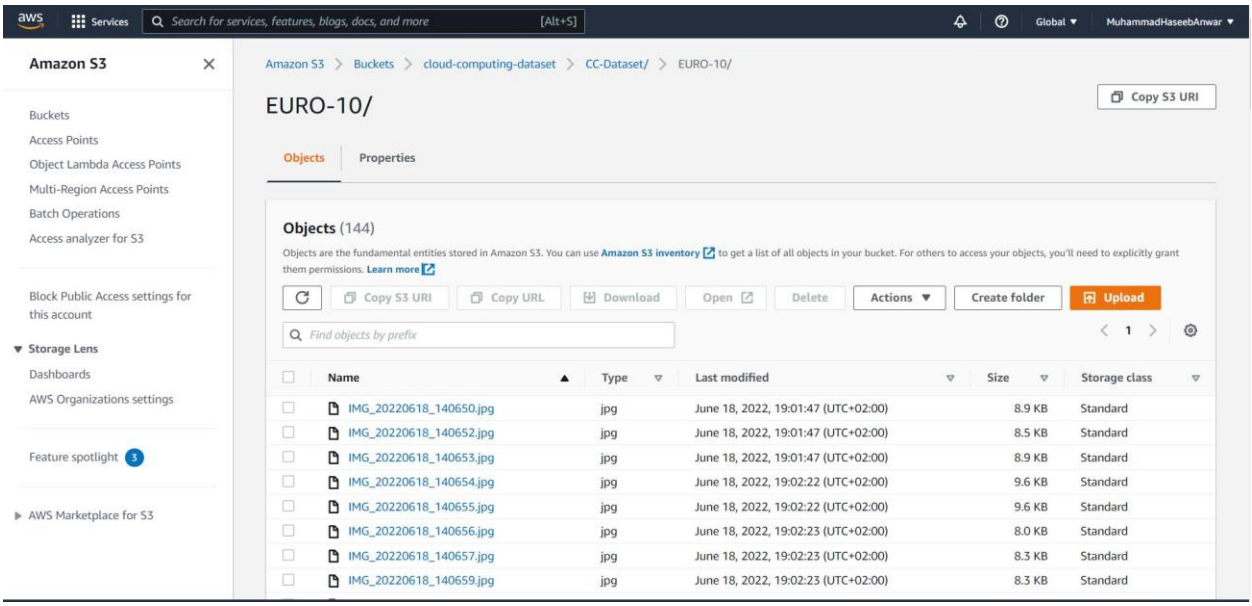


Now, we need to gather our training data, for this model we have taken around 150 sample images of each 5,10,20 and 50 Euro currency notes.

Create folder for each label as shown in picture below.



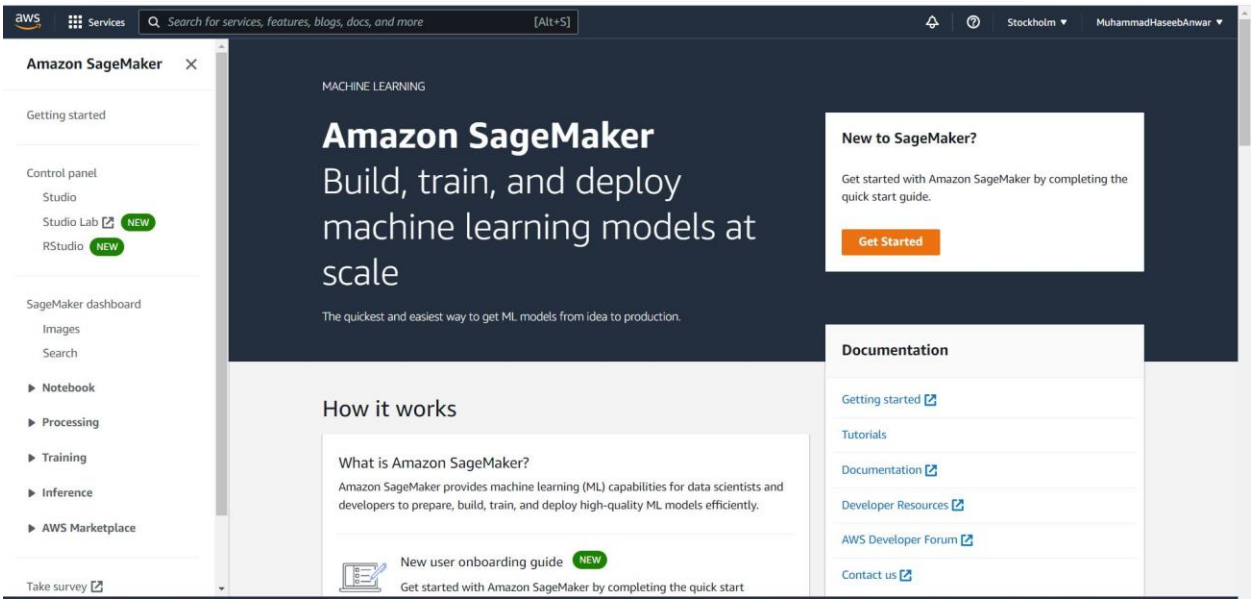
Now upload sample images in respective folders.



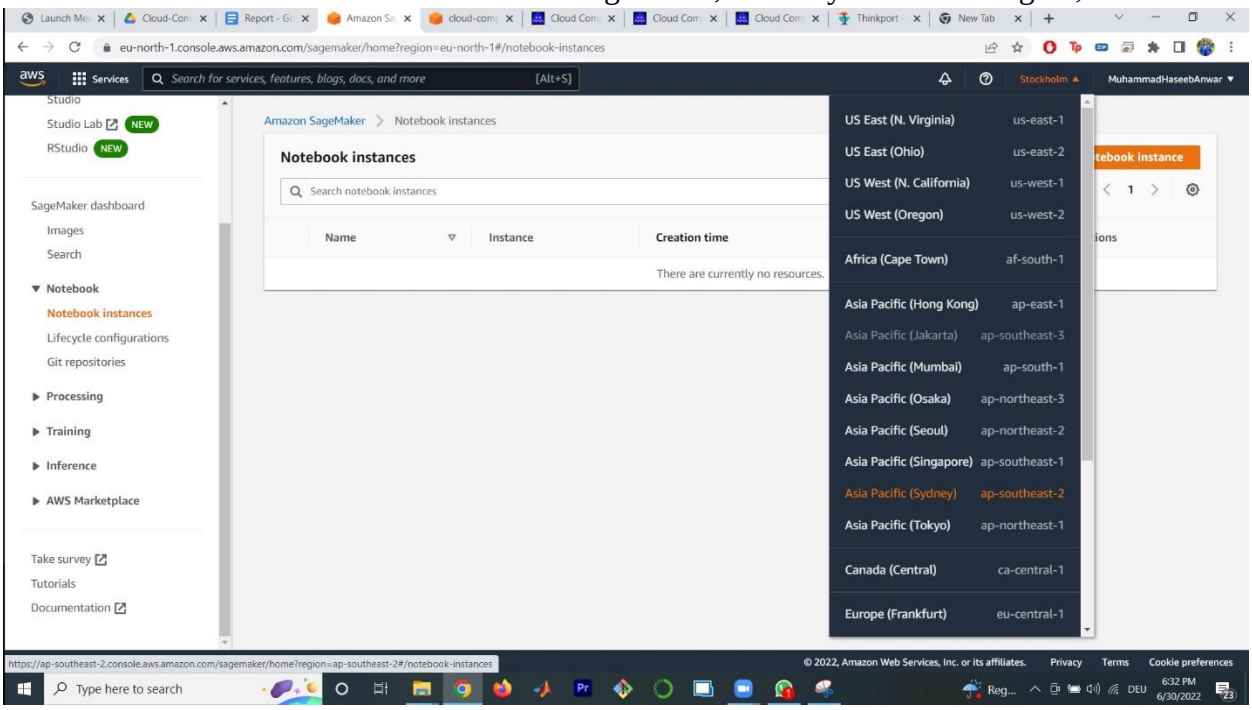
3.1.3 Creating notebook instance on sagemker

Now as we have setup our S3 bucket and folders as we need to train our model, now we are moving towards our AWS service called Sagemaker.

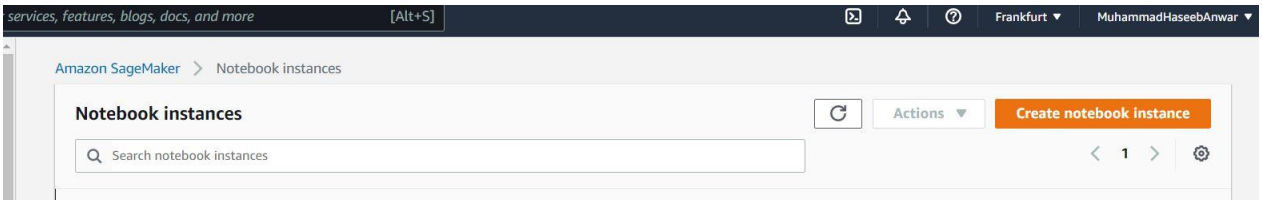
This is the landing page of our Sagemaker.



We need to create Notebook instance within sagemaker, choose your desired region, we have chosen eu-central-

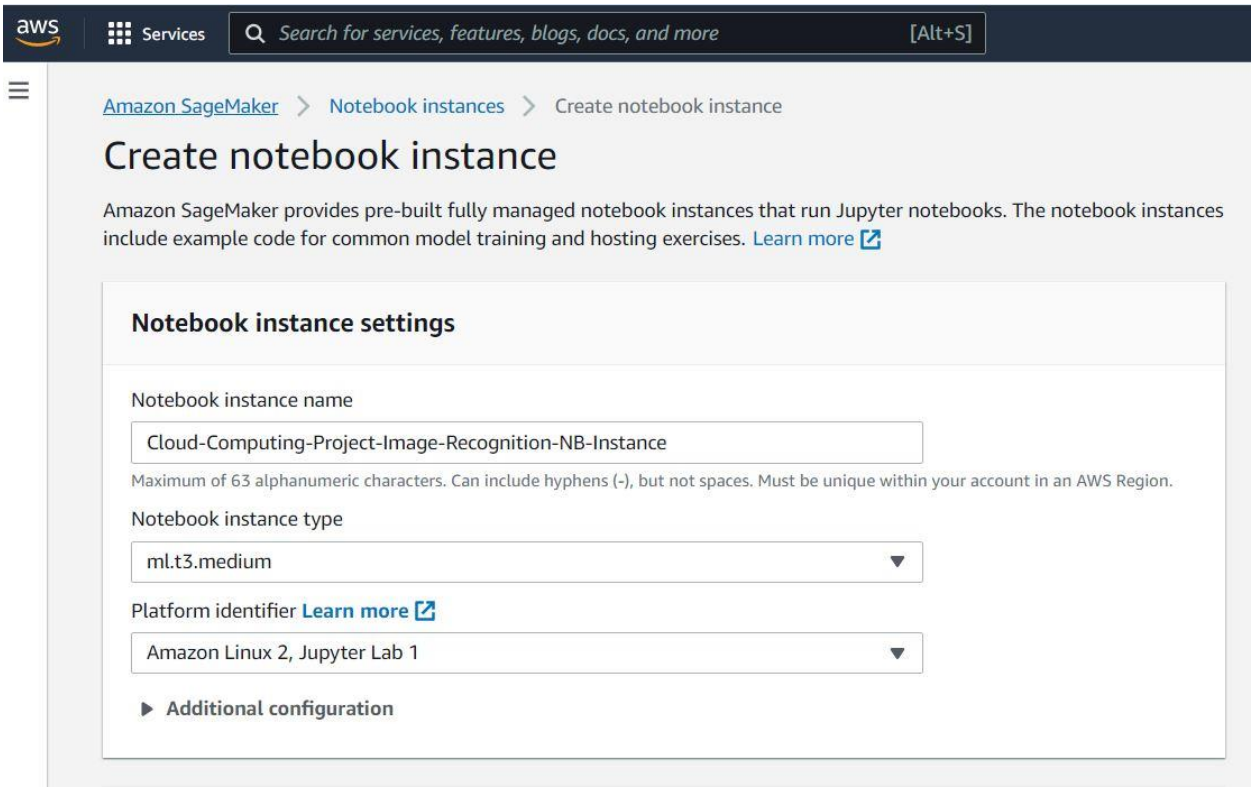


Click on create notebook instance



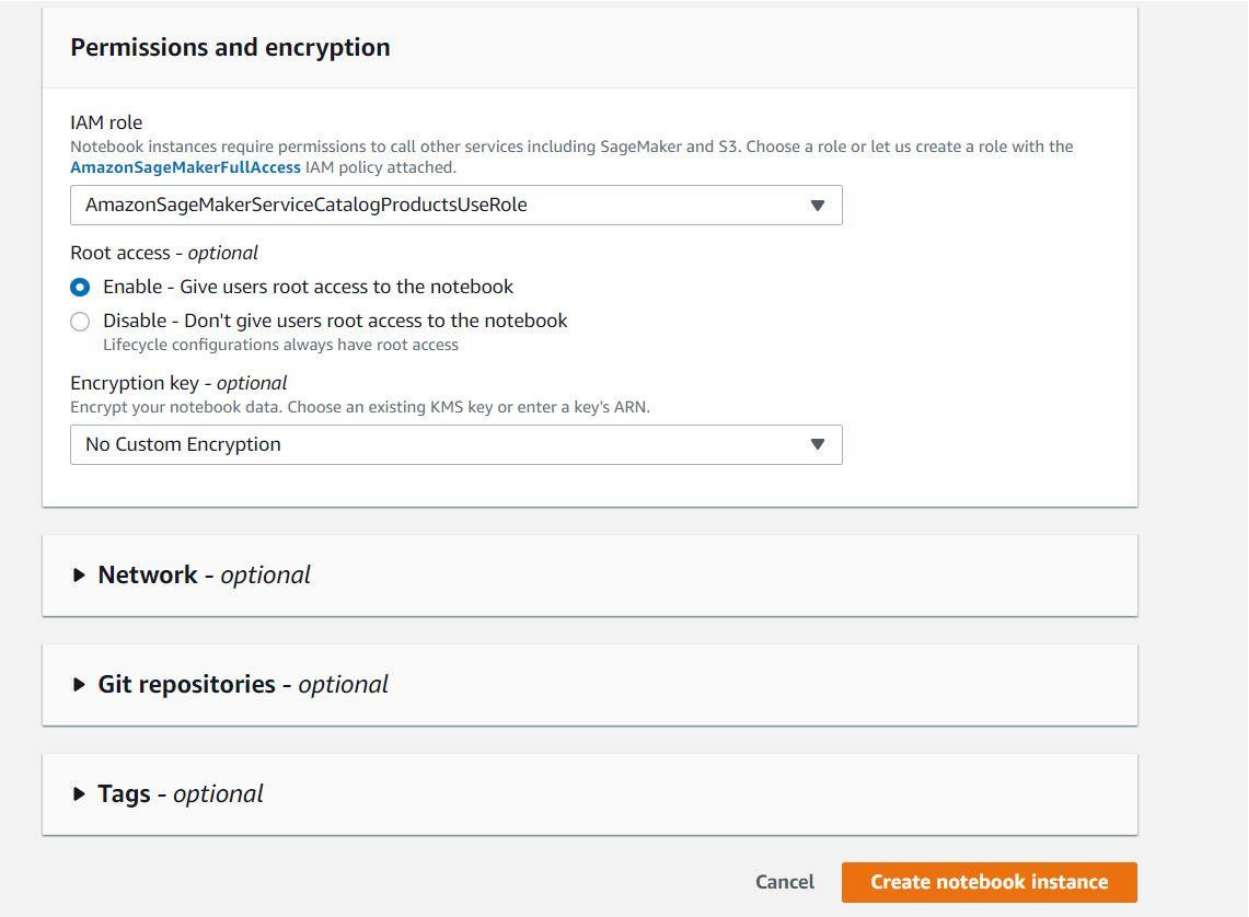
Give name of your notebook instance, and choose your instance type. Further information on sage maker instance types is provided here https://AWS.amazon.com/sagemaker/pricing/ .

If you are using Sagemaker for the first time, then take advantage of the free tier, Free tier information is also provided in the above link.

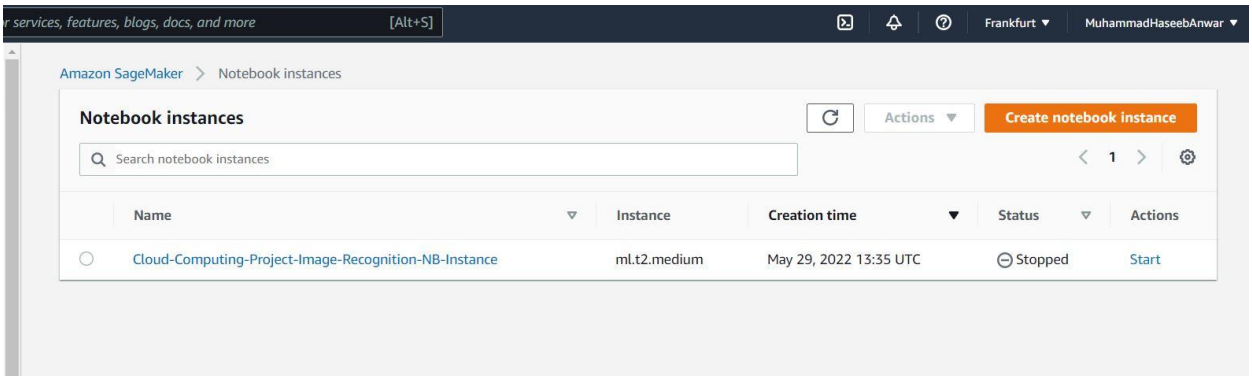


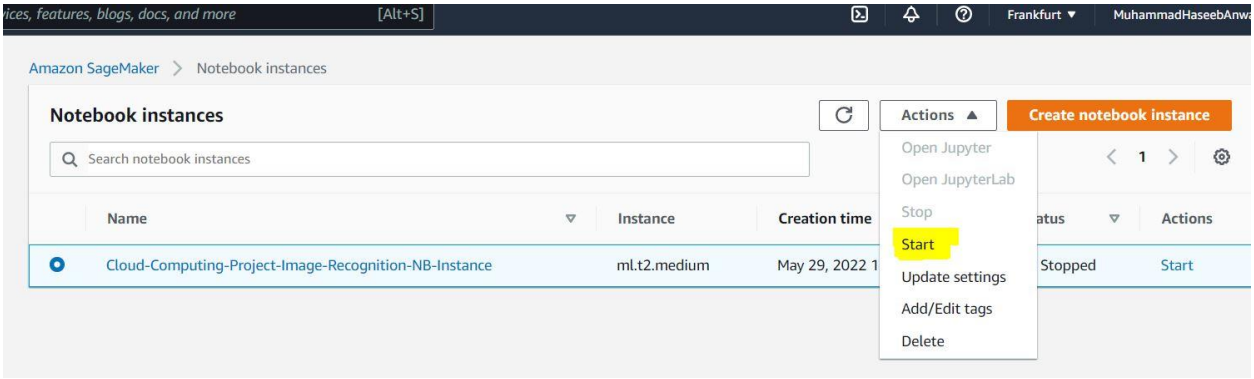
In this step we can create a new IAM role or use already created IAM role. IAM role is nothing but a set of permissions, as you need to access to different other services of AWS within this notebook, make sure that the role which you select here has the necessary permissions to access those web services. Further information of IAM roles are provided here https://docs.AWS.amazon.com/IAM/latest/UserGuide/id_roles.html

Root access: you really don't need to give root access to the sagemaker notebook here if you are working on production environment, root access means administrative privileges, which means by using this notebook you can edit, remove any files on the system.



After our notebook instance is created, we need to start this notebook instance after that we are going to write our model.





3.1.4 Image classification on Sagemaker

Sagemaker Model:

We have used **Sagemaker's built in Image classification Algorithm** which is based on **Supervised Learning** that supports multi-class classification. It uses Conventional Neural Networks (CNN). More info on the image classification model of sagemaker can be access using this link https://docs.AWS.amazon.com/sagemaker/latest/dg/image-classification.html

There are different type of algorithms are offered by sagemaker, which can be used according to the need. Information of different type of algorithms Sagemaker can be accessed here. https://docs.AWS.amazon.com/sagemaker/latest/dg/algos.html

Writing our model:

First we are saving our S3 bucket and main folder name into variables to access again time to time.

[12]:
```
###################### CLOUD COMPUTING PROJECT ############################
######### Training machine learning models in the AWS Sagemaker ############
######### Supervised by: Prof. Dr. Christian Baun ##########################

######### Team Members:
######### Muhammad Haseeb Anwar
######### Moeez Ur Rehman
######### Sehrish Kanwal
######### Harmain Haidar

# S3 Bucket Name
bucket_name='cloud-computing-dataset'

# Our Main Folder inside the S3 bucket Which has subfolders of our classes
# One Sub-Folder will be considered as One Class
dataset_name = 'CC-Dataset'

print('Name of the bucket is: '+dataset_name)
print('Name of dataset folder is: '+bucket_name)
```

```
Name of the bucket is: CC-Dataset
Name of dataset folder is: cloud-computing-dataset
```

Here we are importing the sagemaker library and setting up environment.

Get_execution_role() *method gives us the role which we are using to run the sagemaker notebook.*

session() *method is used to get a sagemaker session.*

Image_uris.retrieve() method is used for generating ECR image URIs for pre-built SageMaker Docker image, the arguments of the methods can be studied extensively using below link. https://sagemaker.readthedocs.io/en/stable/api/utility/image_uris.html#sagemaker.image_uris.retrieve

As we are using prebuilt image classification model of sagemaker we have passed this algorithm name in the perimeter.

[13]:
```
#Setting Up Our Environment

# Importing Sagemaker
# getting execution role of notebook
# defining algorithm type in Image_Uri method
import sagemaker
from sagemaker import get_execution_role
from sagemaker.amazon.amazon_estimator import get_image_uri

role = get_execution_role()
session = sagemaker.Session()
#sagemaker.image_uris.retrieve
#get_image_uri
image_uris = sagemaker.image_uris.retrieve(region=session.boto_region_name, framework ='image-classification')

print('Region name: '+session.boto_region_name)
print('Algorithm Used: image-classification ')
```

```
Region name: eu-central-1
Algorithm Used: image-classification
```

There are multiple ways to feed images to the model for training. The SageMaker Image Classification algorithm supports both RecordIO and conventional image formats like JPG and JPEG. In this project we are going to use the RecordIO format for training.

What is Record IO format?

Data loading is a critical component of any machine learning system. With smaller number of training images, it might not be a problem to used them as they are, but with larger datasets, data loading into training model can become performance critical.

In simple words, Record IO format converts images into binary data exchange formats, RecordIO is efficient data format developed by Apache MXnet it resizes the image into 256 * 256, then compress into JPEG format. After that, it saves a header that indicates the index and label for that image to be used when constructing the *Data* field for that record. It then pack several images together into a file.

More information of RecordIO file can be read here:
https://mxnet.apache.org/versions/1.9.1/api/architecture/note_data_loading

Sagemaker recommend storing images as records and packing them together, the major benefit is Storing images in RecordIO format greatly reduces the size of the dataset on the disk.

In below code we specify the path of the script which converts images into RecordIO files

```
BASE_DIRECTORY='/tmp'

%env BASE_DIRECTORY=$BASE_DIRECTORY
%env S3_BUCKET_NAME = $bucket_name
%env DATASET_NAME = $dataset_name

import sys,os

suffix='/mxnet/tools/im2rec.py'
im2rec = list(filter( (lambda x: os.path.isfile(x + suffix )), sys.path))[0] + suffix
%env IM2REC=$im2rec
```

```
env: BASE_DIR=/tmp
env: S3_DATA_BUCKET_NAME=cloud-computing-dataset
env: DATASET_NAME=CC-Dataset
env: IM2REC=/home/ec2-user/anaconda3/envs/mxnet_p36/lib/python3.6/site-packages/mxnet/tools/im2rec.py
```

As we have specified the script which transforms our images into Record IO file, we now pull all S3 images.

```
[8]: # The script below Pulls our images from S3 bucket

!aws s3 sync s3://$S3_BUCKET_NAME/$DATASET_NAME $BASE_DIRECTORY/$DATASET_NAME --quiet

print('Images have been Pulled!!! ')
```

```
Images have been Pulled!!!
```

Now we transform our fetched images into Record IO file, we have kept the training ratio to 70%, while Testing ratio to 30%. The Record IO files will be created in this step with the above ratio.

[13]:
```
%%bash
# Now here we use the IM2REC script to convert our images which we fetched from S3 bucket into RecordIO files

# Delete if there are already created Recio files in our working directory

cd $BASE_DIR
rm *.rec
rm *.lst

# We want to create 2 LST files first, One for training and One for testing, along with saving the class of each image
# The output of the LST files command includes a list of all of our label classes
# We are specifying here the training and testing ration, 70% Training Ratio and 30% Testing Ratio

echo "Creating LST files"
python $IM2REC --list --recursive --pass-through --test-ratio=0.3 --train-ratio=0.7 $DATASET_NAME $DATASET_NAME > ${DATASET_NAME}_classes

echo "Label classes:"
cat ${DATASET_NAME}_classes

# Then we create RecordIO files from the LST files
echo "Creating RecordIO files"
python $IM2REC --num-thread=4 ${DATASET_NAME}_train.lst $DATASET_NAME
python $IM2REC --num-thread=4 ${DATASET_NAME}_test.lst $DATASET_NAME
ls -lh *.rec
```

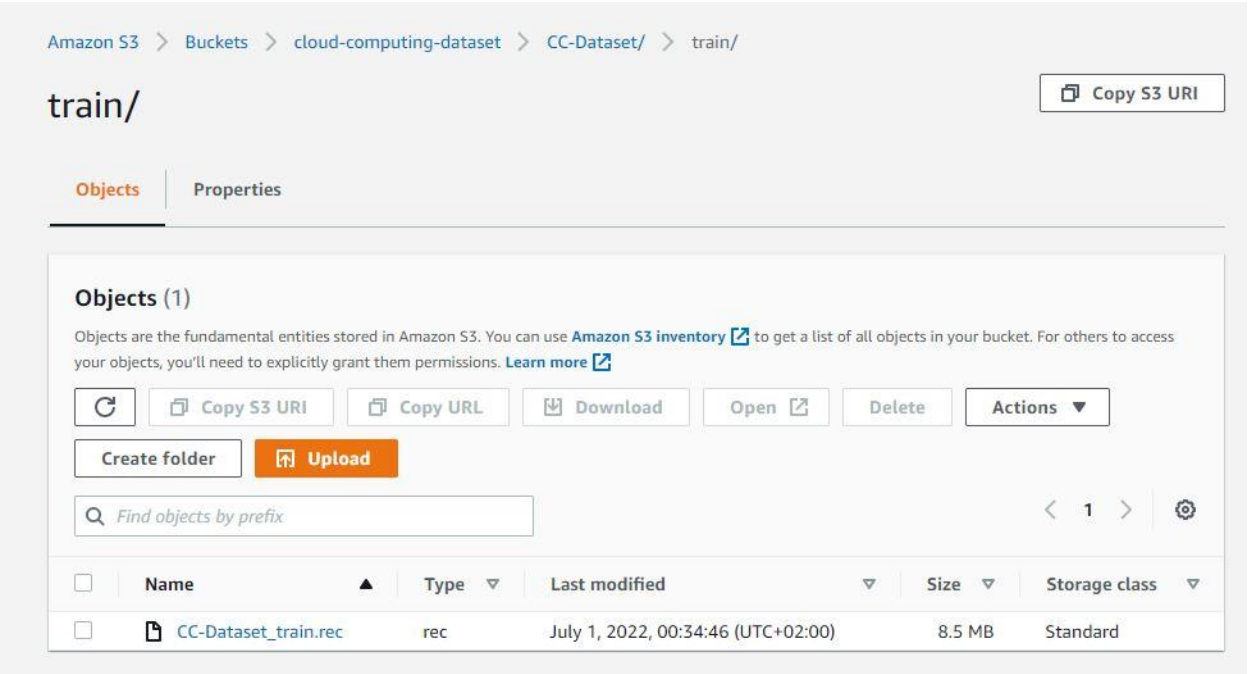
```
Creating LST files
Label classes:
EURO-10 0
EURO-20 1
EURO-5 2
EURO-50 3
Creating RecordIO files
Creating .rec file from /tmp/CC-Dataset_train.lst in /tmp
time: 0.39785265922546387  count: 0
Creating .rec file from /tmp/CC-Dataset_test.lst in /tmp
time: 0.004105329513549805  count: 0
-rw-rw-r-- 1 ec2-user ec2-user 3.7M Jun 30 22:14 CC-Dataset_test.rec
-rw-rw-r-- 1 ec2-user ec2-user 8.6M Jun 30 22:14 CC-Dataset_train.rec
```

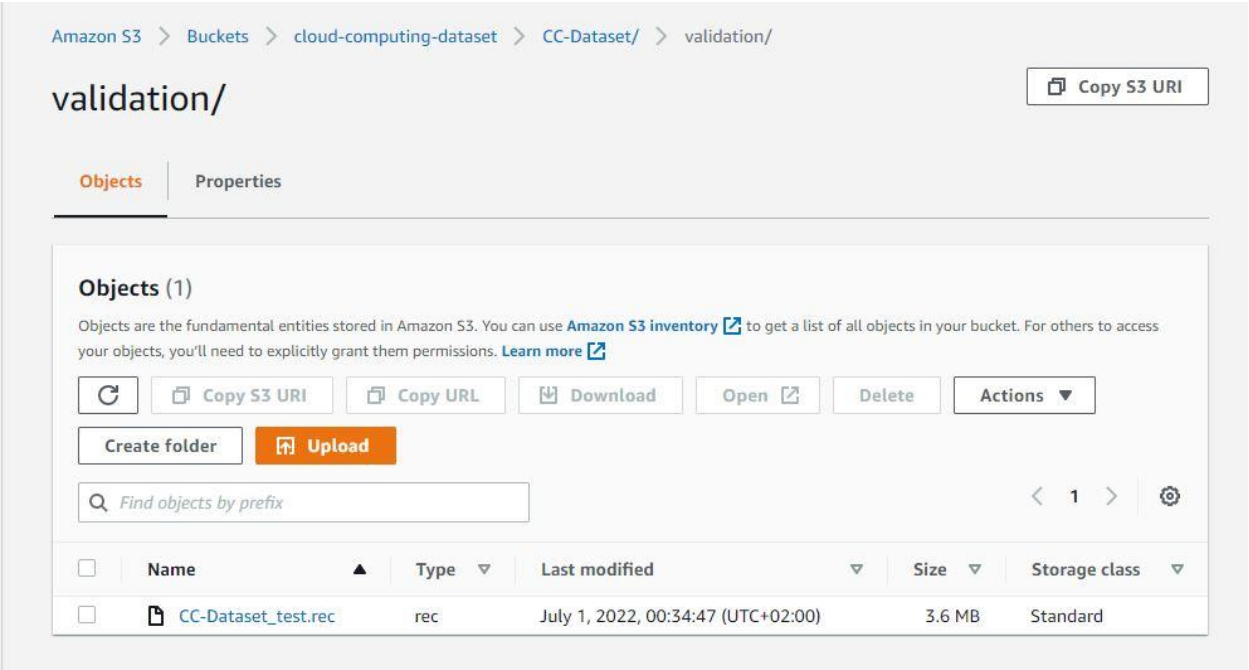
Now we upload our created RecordIO files back into our S3 bucket, which then be used as an input for training of our model.

[15]:
```
# We are now Uploading our train and test RecordIO files to S3 bucket

bucket = bucket_name

print (bucket)

training_path_s3 = 's3://{}/{}/train/'.format(bucket, dataset_name)
validation_path_S3 = 's3://{}/{}/validation/'.format(bucket, dataset_name)
print(training_path_s3)
print(validation_path_S3)

# Delete any existing data
!aws s3 rm s3://{bucket}/{dataset_name}/train --recursive
!aws s3 rm s3://{bucket}/{dataset_name}/validation --recursive

# Upload the rec files to the train and validation folders
!aws s3 cp /tmp/{dataset_name}_train.rec $training_path_s3
!aws s3 cp /tmp/{dataset_name}_test.rec $validation_path_S3
```

```
cloud-computing-dataset
s3://cloud-computing-dataset/CC-Dataset/train/
s3://cloud-computing-dataset/CC-Dataset/validation/
upload: ../../../tmp/CC-Dataset_train.rec to s3://cloud-computing-dataset/CC-Dataset/train/CC-Dataset_train.rec
upload: ../../../tmp/CC-Dataset_test.rec to s3://cloud-computing-dataset/CC-Dataset/validation/CC-Dataset_test.rec
```

The uploaded RecordIO files in our S3 bucket will look like this.





We have now done our preprocessing; the data is ready to be trained. Now are going towards the process of training our model using the created Record IO files.

We are here defining the Record IO paths to the training and validation functions. For information of the inputs. Training Input() method can be found here:
https://sagemaker.readthedocs.io/en/stable/api/utility/inputs.html#sagemaker.inputs.TrainingInput

[20]:
```
# Documentation of the function sagemaker.inputs.TrainingInput is available here
# https://sagemaker.readthedocs.io/en/stable/api/utility/inputs.html#sagemaker.inputs.TrainingInput
# Create a definition for input data used by an SageMaker training job.

train_data = sagemaker.inputs.TrainingInput(
    training_path_s3,
    distribution='FullyReplicated',
    content_type='application/x-recordio',
    s3_data_type='S3Prefix'
)

validation_data = sagemaker.inputs.TrainingInput(
    validation_path_S3,
    distribution='FullyReplicated',
    content_type='application/x-recordio',
    s3_data_type='S3Prefix'
)

data_channels = {'train': train_data, 'validation': validation_data}

print(train_data)
print(validation_data)
```

```
<sagemaker.inputs.TrainingInput object at 0x7f813d49e2e8>
<sagemaker.inputs.TrainingInput object at 0x7f813d49e320>
```

Defining the output location of out model, as well as initializing the estimator function. Sagemaker handles end-to-end Amazon Sagemaker training and deployment tasks. More documentation can be read https://sagemaker.readthedocs.io/en/stable/api/training/estimators.html

[43]:
```
# The are defining the output location for model

s3_output_location = 's3://{}/{}/output'.format(bucket, dataset_name)

# we have used ml.p3.2xlarge isntance for traning
image_classifier = sagemaker.estimator.Estimator(
    role=role,
    image_uri=image_uris,
    instance_count=1,
    instance_type='ml.p3.2xlarge',
    output_path=s3_output_location,
    sagemaker_session=session
)
print('done')
```

```
done
```

Image classification Hyperparameters,

we have defined the

image shape as 3,244,244 which is same as the image shape of our RecordIO files.

number of classes which in our case are 4,

Augmentation type here is important as we are taking the color into account, so we have chosen ‘crop color’.

Epoch: We have not provided any value for epochs so it will take the default value 30.

Learning rates: The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs. The valid values are between 0 and 1.

more documentation can be found here https://docs.AWS.amazon.com/sagemaker/latest/dg/IC-Hyperparameter.html

[58]:
```
num_classes=! ls -l {base_dir}/{dataset_name} | wc -l
num_classes=int(num_classes[0]) - 3
num_training_samples=! cat {base_dir}/{dataset_name}_train.lst | wc -l
num_training_samples = int(num_training_samples[0])

# Details on Sagemaker built-in Image Classifier hyperparameters
# available here: https://docs.aws.amazon.com/sagemaker/latest/dg/IC-Hyperparameter.html

base_hyperparameters=dict(
    use_pretrained_model=1,
    image_shape='3,224,224',
    num_classes=num_classes,
    augmentation_type='crop_color', #taking corresponding Hue-Saturation-Lightness into account
    num_training_samples=num_training_samples,
)

# These are hyperparameters are important which can affect the model training success:
hyperparameters={
    **base_hyperparameters,
    **dict(
        learning_rate=0.001,
        mini_batch_size=5,
    )
}

image_classifier.set_hyperparameters(**hyperparameters)

hyperparameters
print('No of tranining Samples: '+str(num_training_samples))
print ('No of Classes: '+str(num_classes))
```

```
No of tranining Samples: 464
No of Classes: 4
```

Now starting our training job, we have given all our parameters as an input to our training job.

The training job is started and will provide the path of the model where it will be stored.

```
[*]: %%time

import time
now = str(int(time.time()))
training_job_name = 'IC-' + dataset_name.replace('_', '-') + '-' + now

image_classifier.fit(inputs=data_channels, job_name=training_job_name, logs=True)

job = image_classifier.latest_training_job
model_path = f"{BASE_DIR}/{job.name}"

print(f"\n\n Finished training! The model is available for download at: {image_classifier.output_path}/{job.name}/output/model.tar.gz")
```

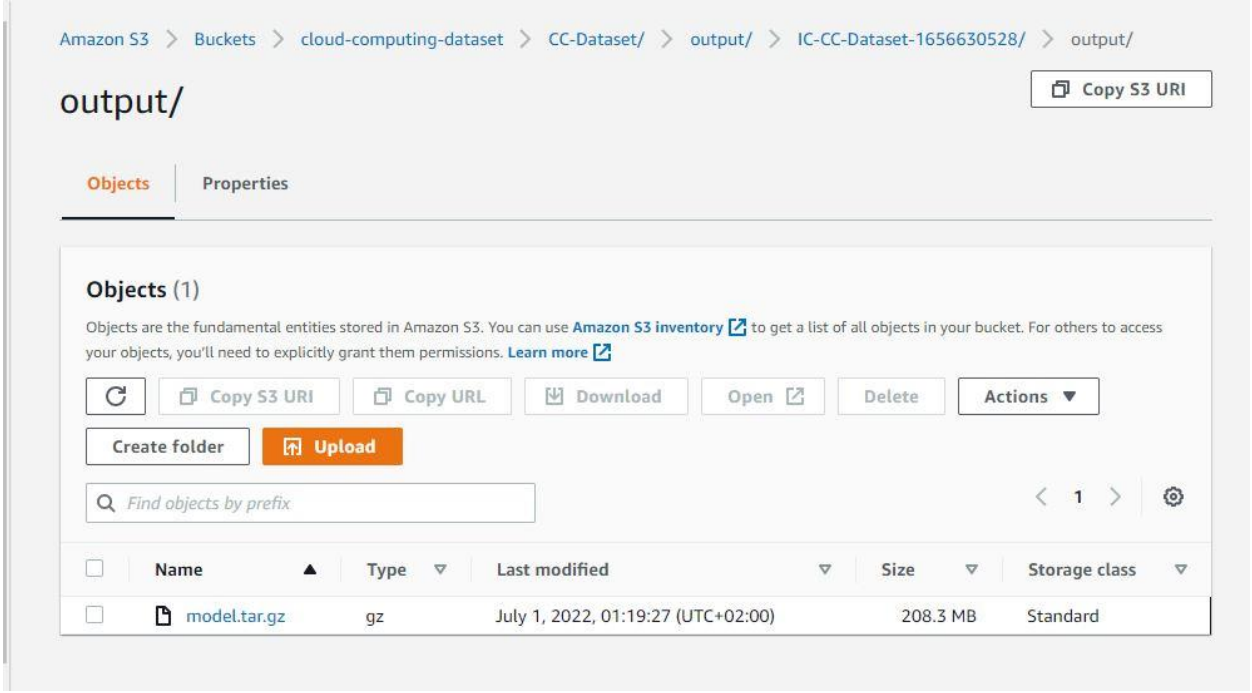
```
2022-06-30 23:08:48 Starting - Starting the training job...ProfilerReport-1656630528: InProgress
...
2022-06-30 23:09:40 Starting - Preparing the instances for training......
2022-06-30 23:10:43 Downloading - Downloading input data..
```

```
2022-06-30 23:13:04 Training - Training image download completed. Training in progress.[23:13:11] /opt/brazil-pkg-cache/packages/AIAlgorithmsMXNet/AIAlgorithmsMXNet-1.3.x_ecl_Cuda_10.1.x.11282.0/AL2_x86_64/generic-flavor/src/src/operator/nn/./cudnn/./cudnn_algoreg-inl.h:97: Running performance tests to find the best convolution algorithm, this can take a while... (setting env variable MXNET_CUDNN_AUTOTUNE_DEFAULT to 0 to disable)
[06/30/2022 23:13:15 INFO 140493106341696] Epoch[0] Batch [20]#011Speed: 23.524 samples/sec#011accuracy=0.428571
[06/30/2022 23:13:16 INFO 140493106341696] Epoch[0] Batch [40]#011Speed: 32.955 samples/sec#011accuracy=0.609756
[06/30/2022 23:13:18 INFO 140493106341696] Epoch[0] Batch [60]#011Speed: 38.072 samples/sec#011accuracy=0.708197
[06/30/2022 23:13:20 INFO 140493106341696] Epoch[0] Batch [80]#011Speed: 41.102 samples/sec#011accuracy=0.767901
[06/30/2022 23:13:21 INFO 140493106341696] Epoch[0] Train-accuracy=0.784783
[06/30/2022 23:13:21 INFO 140493106341696] Epoch[0] Time cost=10.725
[06/30/2022 23:13:22 INFO 140493106341696] Epoch[0] Validation-accuracy=0.995000
[06/30/2022 23:13:23 INFO 140493106341696] Storing the best model with validation accuracy: 0.995000
[06/30/2022 23:13:23 INFO 140493106341696] Saved checkpoint to "/opt/ml/model/image-classification-0001.params"
[06/30/2022 23:13:25 INFO 140493106341696] Epoch[1] Batch [20]#011Speed: 55.178 samples/sec#011accuracy=0.895238
[06/30/2022 23:13:27 INFO 140493106341696] Epoch[1] Batch [40]#011Speed: 55.481 samples/sec#011accuracy=0.902439
[06/30/2022 23:13:28 INFO 140493106341696] Epoch[1] Batch [60]#011Speed: 55.671 samples/sec#011accuracy=0.927869
[06/30/2022 23:13:30 INFO 140493106341696] Epoch[1] Batch [80]#011Speed: 55.449 samples/sec#011accuracy=0.938272
[06/30/2022 23:13:31 INFO 140493106341696] Epoch[1] Train-accuracy=0.936957
[06/30/2022 23:13:31 INFO 140493106341696] Epoch[1] Time cost=8.199
[06/30/2022 23:13:32 INFO 140493106341696] Epoch[1] Validation-accuracy=1.000000
[06/30/2022 23:13:33 INFO 140493106341696] Storing the best model with validation accuracy: 1.000000
[06/30/2022 23:13:33 INFO 140493106341696] Saved checkpoint to "/opt/ml/model/image-classification-0002.params"
[06/30/2022 23:13:35 INFO 140493106341696] Epoch[2] Batch [20]#011Speed: 54.774 samples/sec#011accuracy=0.914286
[06/30/2022 23:13:37 INFO 140493106341696] Epoch[2] Batch [40]#011Speed: 55.316 samples/sec#011accuracy=0.951220
[06/30/2022 23:13:39 INFO 140493106341696] Epoch[2] Batch [60]#011Speed: 55.483 samples/sec#011accuracy=0.963934
[06/30/2022 23:13:40 INFO 140493106341696] Epoch[2] Batch [80]#011Speed: 55.151 samples/sec#011accuracy=0.967901
[06/30/2022 23:13:41 INFO 140493106341696] Epoch[2] Train-accuracy=0.971739
[06/30/2022 23:13:41 INFO 140493106341696] Epoch[2] Time cost=8.231
[06/30/2022 23:13:42 INFO 140493106341696] Epoch[2] Validation-accuracy=1.000000
[06/30/2022 23:13:45 INFO 140493106341696] Epoch[3] Batch [20]#011Speed: 53.828 samples/sec#011accuracy=0.990476
[06/30/2022 23:13:47 INFO 140493106341696] Epoch[3] Batch [40]#011Speed: 55.007 samples/sec#011accuracy=0.990244
[06/30/2022 23:13:48 INFO 140493106341696] Epoch[3] Batch [60]#011Speed: 55.255 samples/sec#011accuracy=0.977049
[06/30/2022 23:13:50 INFO 140493106341696] Epoch[3] Batch [80]#011Speed: 55.150 samples/sec#011accuracy=0.980247
[06/30/2022 23:13:51 INFO 140493106341696] Epoch[3] Train-accuracy=0.976087
[06/30/2022 23:13:51 INFO 140493106341696] Epoch[3] Time cost=8.235
```

```
print(f"\n\n Finished training! The model is available for download at: {image_classifier.output_path}/{job.name}/output/model.tar.gz")
```

```
 Finished training! The model is available for download at: s3://cloud-computing-dataset/CC-Dataset/output/IC-CC-Dataset-1656630528/output/model.tar.gz
```

We can also see our S3 bucket where the model is saved.



3.1.5 Deploying endpoint on sagemaker

As our model is trained, we now have to deploy our endpoint, which then will be used for our predictions.

```
[63]: %%time
# Deploying our trained model to an endpoint which will then we used by our app to predict the currency

deployed_endpoint = image_classifier.deploy(
    initial_instance_count = 1,
    instance_type = 'ml.t2.medium'
)
```

```
---------!CPU times: user 141 ms, sys: 16.2 ms, total: 157 ms
Wall time: 4min 31s
```

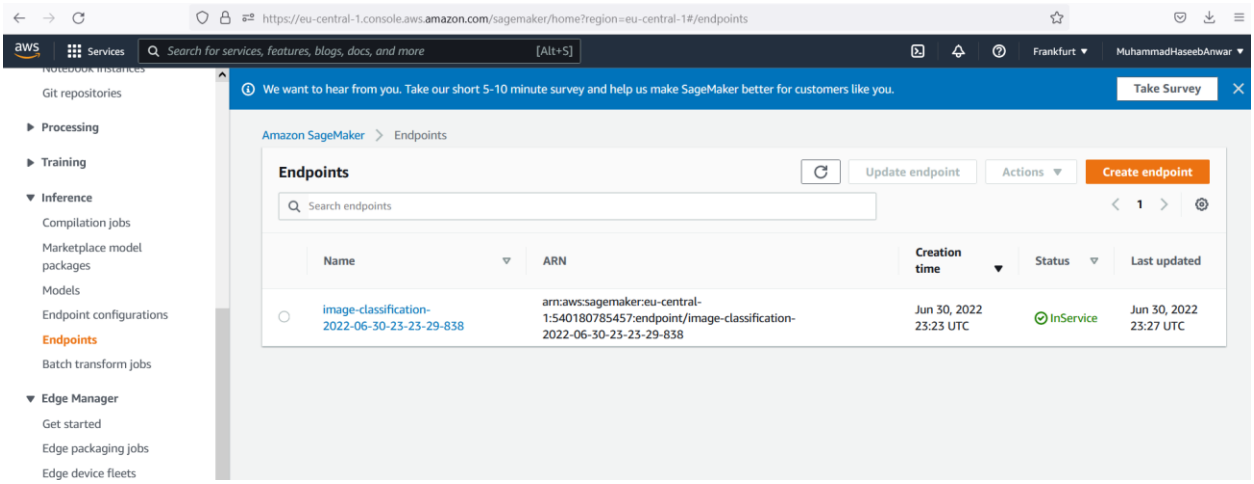
```
[64]: print(deployed_endpoint)
```

```
<sagemaker.predictor.Predictor object at 0x7f813cab0d30>
```

The deployed endpoint is available and in service now.

Now there are many ways to call this endpoint, we can predict our images from this notebook or we can create a webapp which will call our endpoint using Api Gateway and Lambda functions and will show our predictions, first we are showing how this can be done using notebook instance.

Create a function which invokes the endpoint and returns the result of prediction.

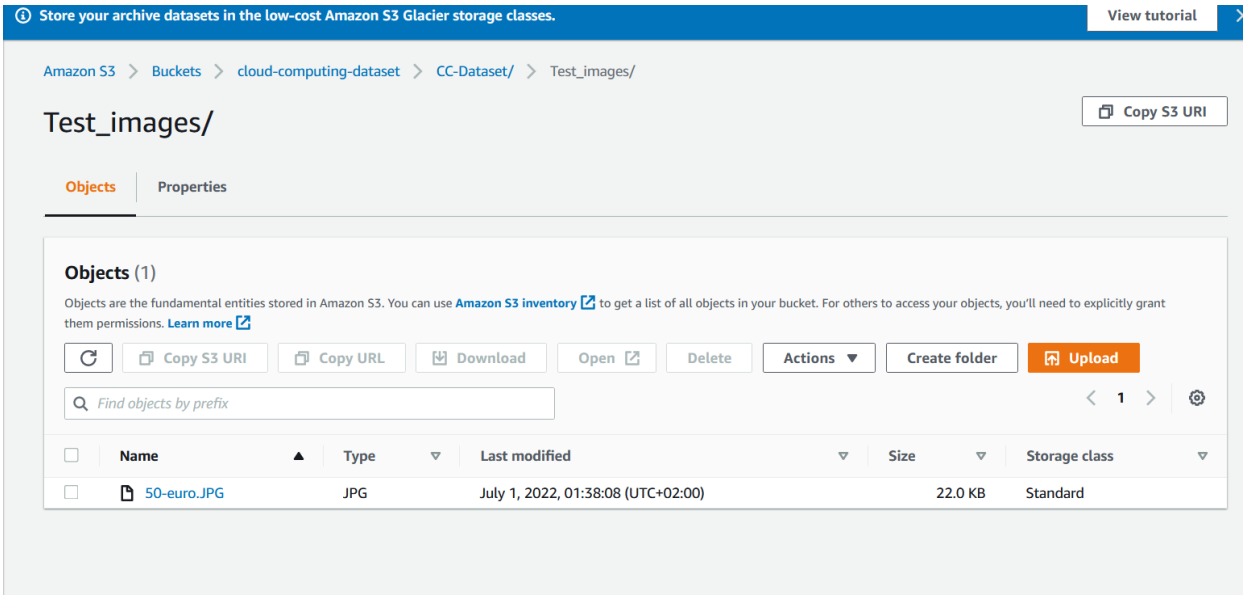


3.1.6 Testing the model from notebook

[66]:
```
# If we want to check out model' prediction through this notebook instance
# we will create a function which will call our endpoint here and return the model prediction
# we will have to upload some test images to our s3 bucket which this method will use

import json
import numpy as np
import os

def classify_deployed(file_name, classes):
    payload = None
    with open(file_name, 'rb') as f:
        payload = f.read()
        payload = bytearray(payload)

    result = deployed_endpoint.predict(payload, initial_args={'ContentType': 'image/jpeg'})

    #result = json.loads(deployed_endpoint.predict(payload))
    #result = deployed_endpoint.predict(payload)
    #best_prob_index = np.argmax(result)
    #return (classes[best_prob_index], result[best_prob_index])

    resultarray = (result.decode('UTF-8')[1:len(result)-1]).split(",")

    for i in range(len(classes)):
        print(classes[i] + ":" + str(resultarray[i]))
    return result


print("Function created")
```

```
Function created
```

Now upload a sample image into S3 Bucket, We have used a 50 euro image and uploaded in into the folder test_images on S3.



In this step we are getting our image from S3 and saving them into a variable called Euro50.

```
[68]: # but for this we have to make our image public

!wget -O test.jpg https://cloud-computing-dataset.s3.eu-central-1.amazonaws.com/CC-Dataset/Test_images/50-euro.JPG
Euro50 = "test.jpg"
# test image
from IPython.display import Image

Image(Euro50)
```

```
--2022-06-30 23:39:43--  https://cloud-computing-dataset.s3.eu-central-1.amazonaws.com/CC-Dataset/Test_images/50-euro.JPG
Resolving cloud-computing-dataset.s3.eu-central-1.amazonaws.com (cloud-computing-dataset.s3.eu-central-1.amazonaws.com)... 52.219.47.144
Connecting to cloud-computing-dataset.s3.eu-central-1.amazonaws.com (cloud-computing-dataset.s3.eu-central-1.amazonaws.com)|52.219.47.144|:4
43... connected.
HTTP request sent, awaiting response... 200 OK
Length: 22513 (22K) [image/jpeg]
Saving to: 'test.jpg'

test.jpg            100%[===================>]  21.99K  --.-KB/s    in 0.001s

2022-06-30 23:39:44 (21.8 MB/s) - 'test.jpg' saved [22513/22513]
```

[68]:

Now we are calling our classify_deployed function to predict our image, and the result is shown in below image.

```
43... connected.
HTTP request sent, awaiting response... 200 OK
Length: 22513 (22K) [image/jpeg]
Saving to: 'test.jpg'

test.jpg            100%[===================>]  21.99K  --.-KB/s    in 0.001s

2022-06-30 23:39:44 (21.8 MB/s) - 'test.jpg' saved [22513/22513]
```

[68]:

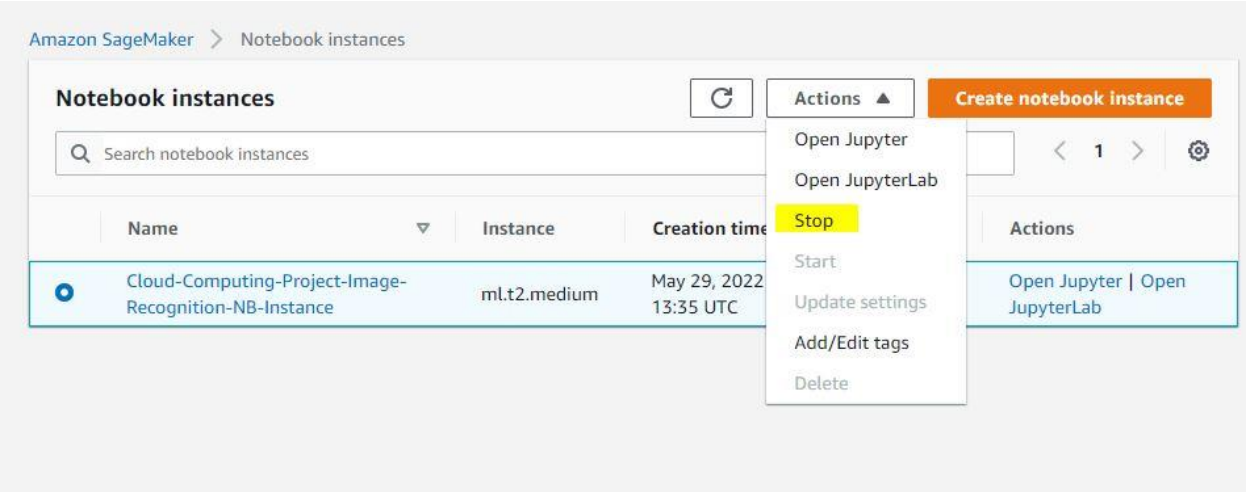
[70]:
```
object_categories = [
    "EURO-10",
    "EURO-5",
    "EURO-20",
    "EURO-50"
]

output = classify_deployed(Euro50,object_categories)
```

```
EURO-10:2.640426464495249e-05
EURO-5: 4.2157054849667475e-05
EURO-20: 9.781115659279749e-05
EURO-50: 0.9998335838317871
```

As we can see the EURO-50 image has the highest prediction value, our model is 99.9% confident that the provided image in of 50 Euro note.

3.1.7 Cleanup Sagemaker

Important: Important is to stop the notebook after you have done creating the model, otherwise AWS will keep charging you for the time the notebook is in service.



4. Web Application:

The web application is created for demo purposes of the model deployed on AWS Sagemaker. It is created using React libraries in addition to using amplify library which streamlines the connection of the web application with the AWS cloud setup.

For setting up amplify in the project we need to install amplify and with in the project directory execute the command:

amplify init

Now we need to setup an API endpoint on the AWS API Gateway. We use the command:

amplify add api

We further follow the steps in the process executed by the command to make a POST Rest api. When the api end point is created locally we push the setup on the AWS using:

amplify push

The web application consists of 2 main components:

Image Capture:

This component uses the camera of the device to capture the image to be identified.

```
src > components > JS ImageCapture.js > ImageCapture > render
26        // Moeez comments: This component opens and captures or screenshots
27        // the image in the camera session which is then used for image recognition
28        return (
29          <div>
30            <div>
31              <Webcam
32                audio={false}
33                height={IMAGE_HEIGHT}
34                width={IMAGE_WIDTH}
35                ref={this.setRef}
36                screenshotFormat="image/jpeg"
37                screenshotWidth={IMAGE_WIDTH}
38                videoConstraints={videoConstraints}
39              />
40            </div>
41
42            <Form.Button onClick={this.handleCapture}>Classify</Form.Button>
43          </div>
44        );
45      }
46    }
```

Classified Image:

This component displays the result in a format of a card with header, information and details.

App

```
src > components > JS ClassifiedImage.js > ...
36
37      // Moeez comments: This component creates a React UI Card which is consists of a
38      // Header, Meta and a Description element.
39      render() {
40        return (
41          <Card style={{width: '224px'}}>
42            <Image src={this.props.imageSrc} />
43            <Card.Content>
44              <Card.Header>
45                { this.state.bestLabel ? this.state.bestLabel : "Loading..." }
46              </Card.Header>
47              <Card.Meta>
48                { this.state.bestLabelScore ? this.state.bestLabelScore : "" }
49              </Card.Meta>
50              <Card.Description>
51                <Accordion defaultActiveIndex={-1} panels={this.accordionPanels()} />
52              </Card.Description>
53            </Card.Content>
54          </Card>
55        )
56      }
57    }
58
```

Class:

The main class of the web app is the App.js class where everything is put together.

First we make a form t take the inputs where we want to send the POST request to. The input includes name of the AWS Inference endpoint of Sagemaker, its region and the labels or categories we want to identify or find results of.

App.js M

src > App.js > App > render

```
147          {/* Moeez comments: This is form to take inputs which are used to send the
148          post request to and the labels are used to present the result of specific
149          categories. */}
150        <Form>
151          <Form.Group widths='equal'>
152            <Form.Input label='SM Endpoint Name' placeholder='Please enter Sagemaker endpoint name'
153            name='endpointName' onChange={this.changeHandler} value={this.state.endpointName} />
154
155            <Form.Input label='SM Endpoint Region' placeholder='Please enter sagemaker endpoint region'
156            name='endpointRegion' onChange={this.changeHandler} value={this.state.endpointRegion} />
157
158            <Form.Input label='Labels' placeholder='Please enter space delimited list of labels'
159            name='labels' onChange={this.changeHandler} value={this.state.labels} />
160          </Form.Group>
161
162          <Form.Group widths='equal'>
163            <ImageCapture onCapture={this.classify}/>
164          </Form.Group>
165        </Form>
```

Secondly after capturing image and issuing a POST request to AWS API Gateway endpoint, we receive the response and we display it to the user via a Card Group with our component of ClassifiedImage.

A

App.js M

src > App.js > App > render

```
187
188          {/* Moeez comments: Finally the results are grouped together and displayed */}
189        <CardGroup>
190          { this.state.imageSources.map((src, index) =>
191          <ClassifiedImage key={"img"+index} imageSrc={src} classifier={this.classifier} />) }
192        </CardGroup>
```

snapshot of the request that is sent is attached here:

App.js M

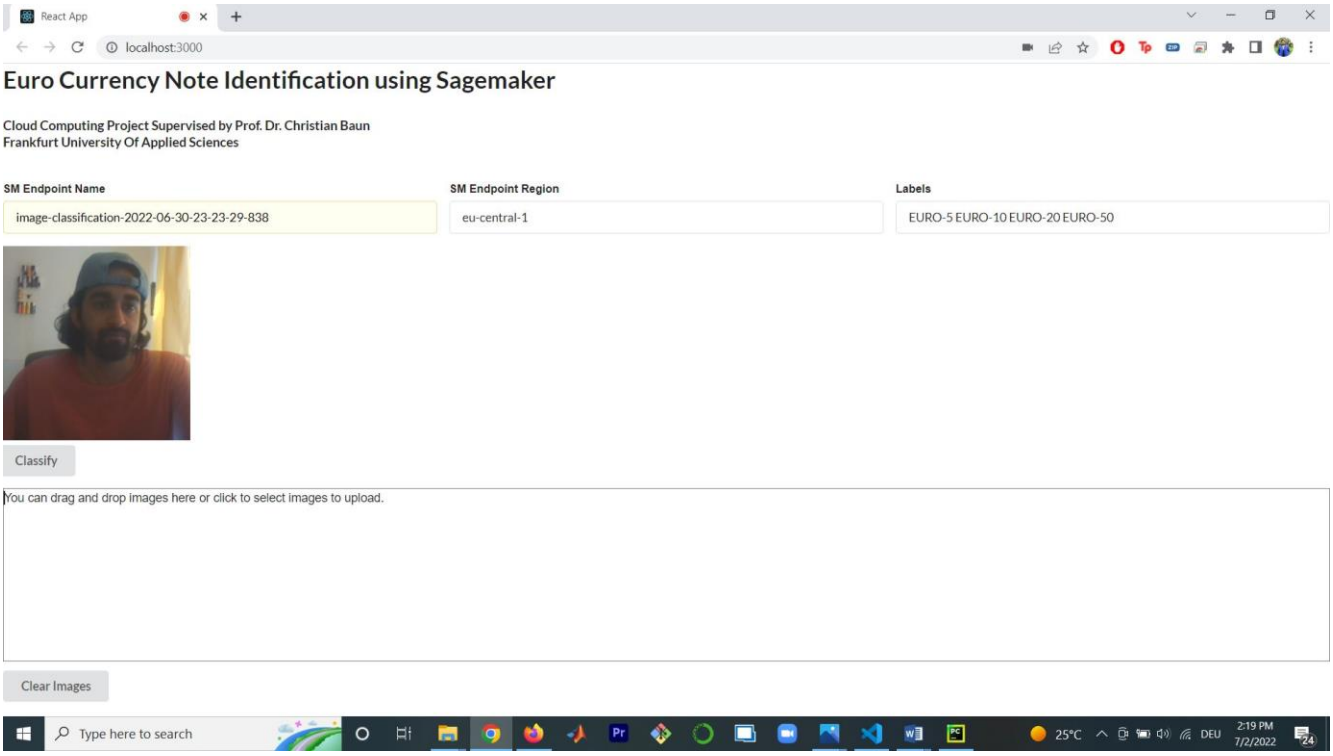
src > App.js > App > render

```
83
84  // Moeez comments: This function calls the AWS API Gateway endpointName
85  // and returns the categories with their predictions as the result
86  classifier = async (imageSrc) => {
87    const base64Image = new Buffer(imageSrc.replace(/^data:image\/\w+;base64,/, ""), 'base64')
88    const { predictions } = await API.post(
89      aws_exports.aws_cloud_logic_custom[0].name,
90      '/classify',
91      {
92        body: {
93          base64Image,
94          endpointName: this.state.endpointName,
95          endpointRegion: this.state.endpointRegion,
96        },
97      }
98    );
99    const topProbIndex = argMax(predictions);
100   const labels = [].concat(this.state.labels.split(' '));
101   labels.sort();
102   return {
103     labels: labels, predictions, topProbIndex: topProbIndex
104   }
105 }
106
```

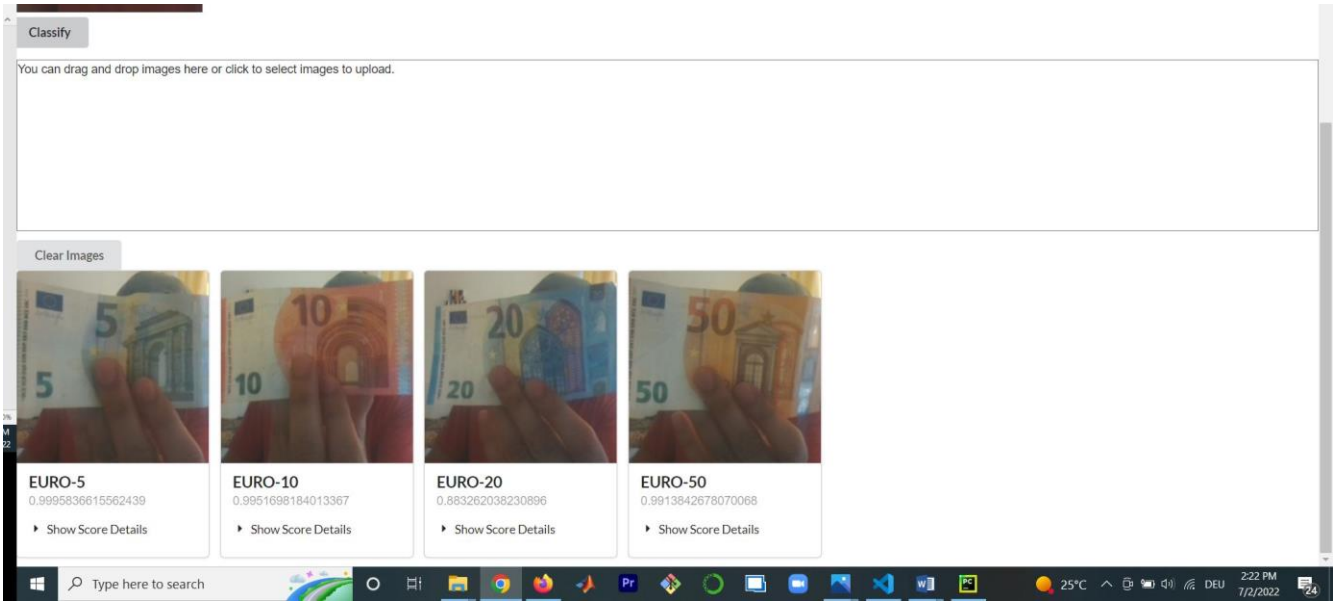
5. Predicting currency note from webapp

Results:

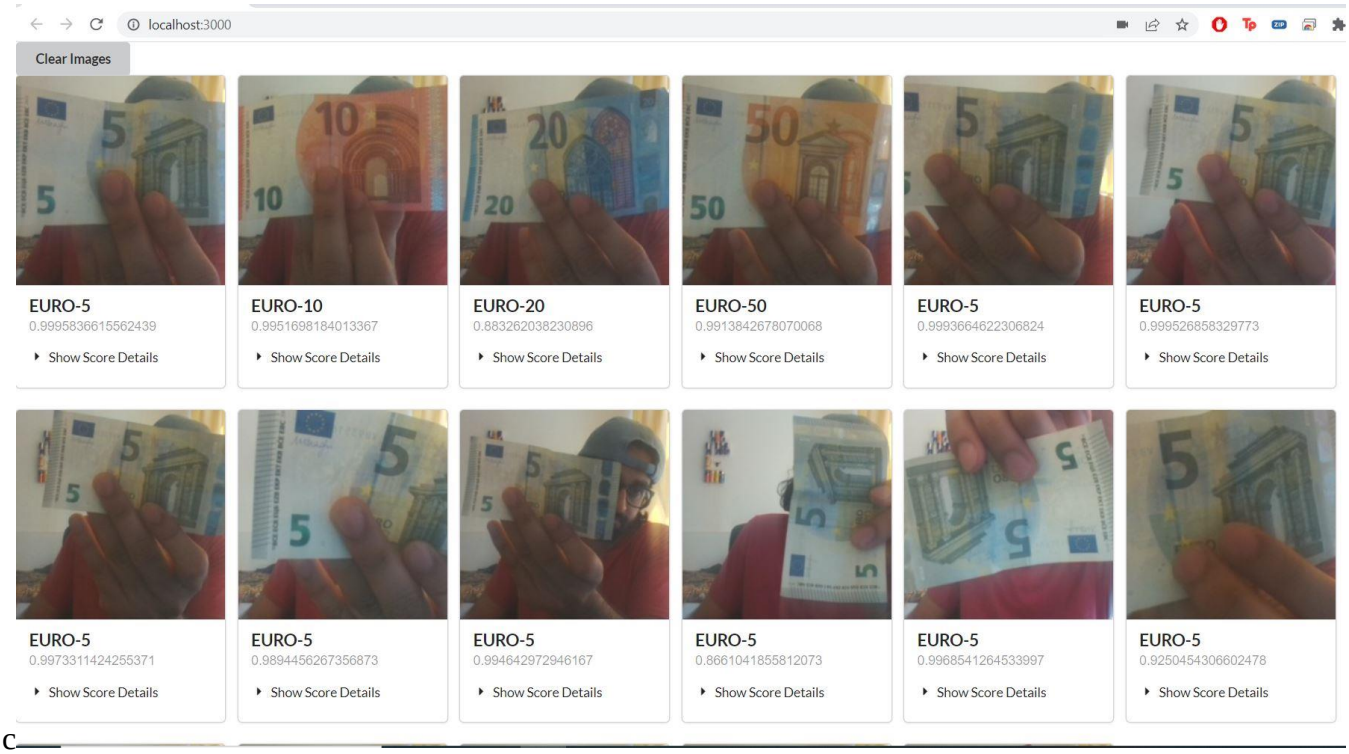
The webpage needs 3 inputs, Sagemaker endpoint name, Region Name and all the classes names with ‘space’



We capture images by pressing the classify button, and then prediction score will be shown below.



We tried to predict the notes with different angles, to see how our prediction is working.



C