Task 5 : Azure Custom Vision Using Python SDK

Abstract

Azure Custom Vision is an image recognition service offered as part of Azure Cognitive Services. It enables users to train custom machine learning models using their own images. There are two main types of models that can be trained: object detection and classification.

Object detection models allow users to identify and locate multiple objects within an image by drawing bounding boxes around them. Classification models, on the other hand, classify images into predefined categories without providing specific location information.

In this project, we will focus on training an object detection model to classify and locate four different types of vehicles within images - Car, Metro, Bus and Ship.

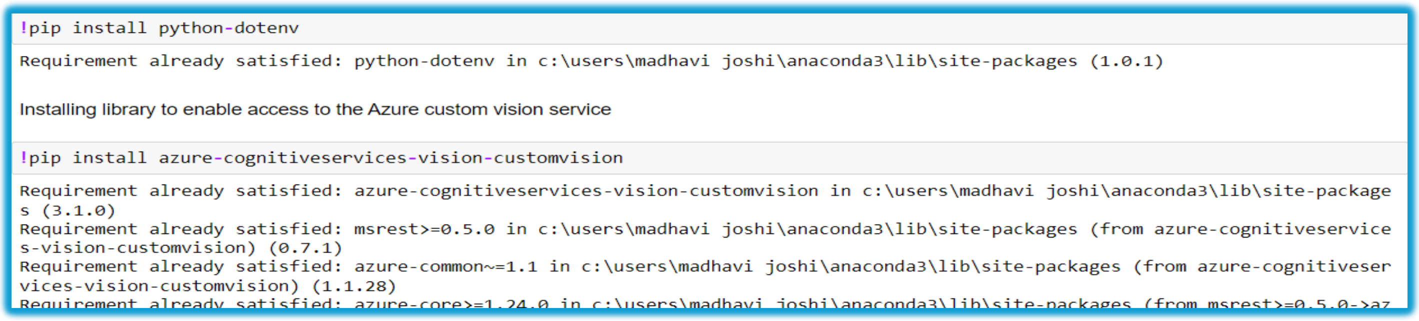
Overview of the steps involved in the project

- **Create Custom Vision Resources**: Set up Custom Vision resources in the Azure Portal to begin training your model.
- **Create a New Project**: Create a new project on the Custom Vision page, which will serve as the workspace for training your model.
- **Choose Training Images and Annotations**: Select and annotate training images, providing bounding box information for each object to be detected.
- **Train the Object Detection Model**: Train your custom object detection model using the annotated training data.
- **Evaluate the Model**: Assess the performance of the trained model using evaluation metrics and validation data.
- **Test Your Model**: Test the trained model by making predictions on new test images to ensure its effectiveness.
- **Publish Model**: Publish the trained model to make it available for deployment and inference.

Finally, you can upload test images to the Custom Vision tool to obtain predictions from the trained model and validate its performance.

Prerequisites

Installing library to read environment variables.



```
!pip install python-dotenv
```

```
Requirement already satisfied: python-dotenv in c:\users\madhavi joshi\anaconda3\lib\site-packages (1.0.1)
```

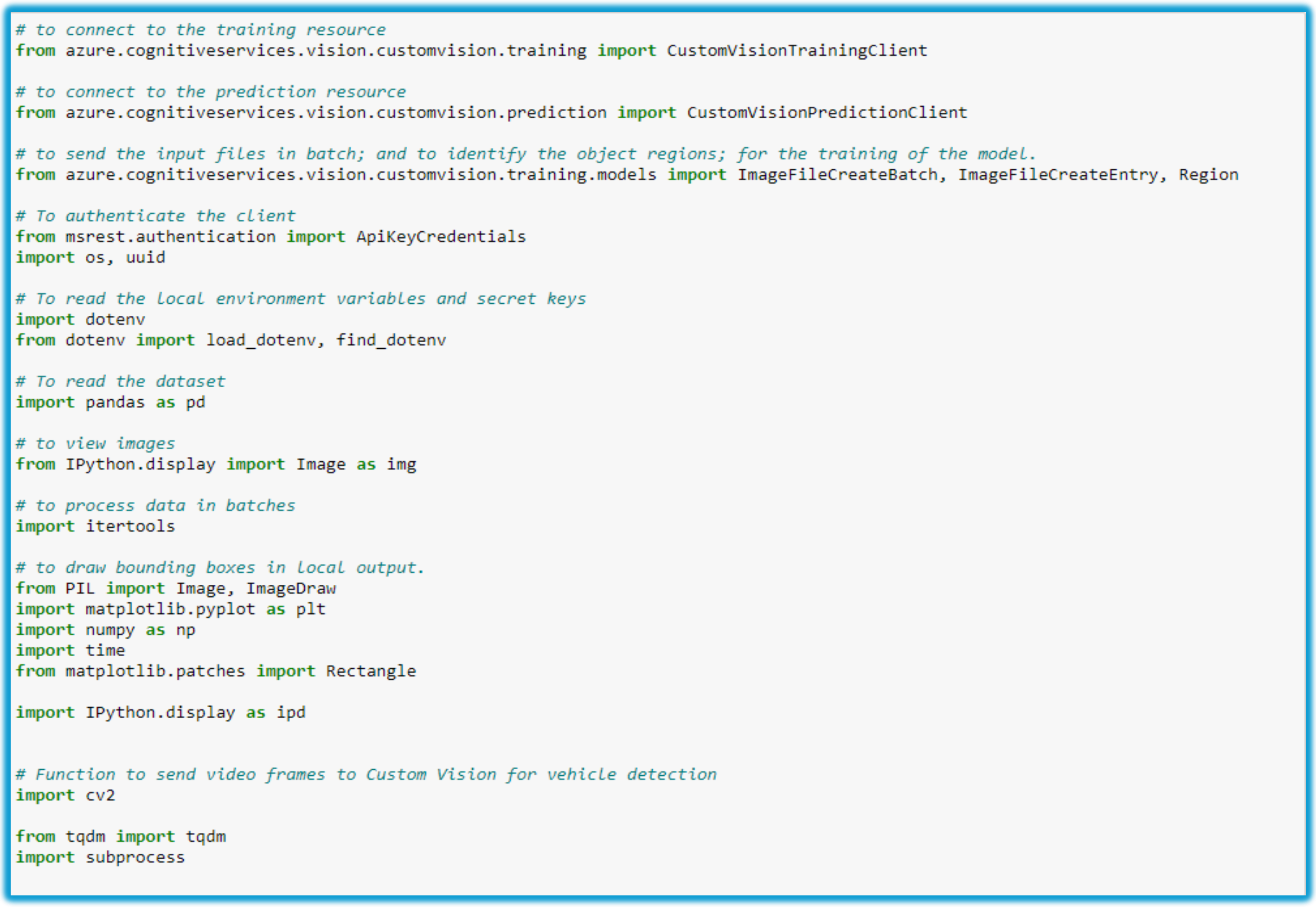
Installing library to enable access to the Azure custom vision service

```
!pip install azure-cognitiveservices-vision-customvision
```

```
Requirement already satisfied: azure-cognitiveservices-vision-customvision in c:\users\madhavi joshi\anaconda3\lib\site-package
s (3.1.0)
Requirement already satisfied: msrest>=0.5.0 in c:\users\madhavi joshi\anaconda3\lib\site-packages (from azure-cognitiveservice
s-vision-customvision) (0.7.1)
Requirement already satisfied: azure-common~=1.1 in c:\users\madhavi joshi\anaconda3\lib\site-packages (from azure-cognitiveser
vices-vision-customvision) (1.1.28)
Requirement already satisfied: azure-core>=1.24.0 in c:\users\madhavi joshi\anaconda3\lib\site-packages (from msrest>=0.5.0->az
```

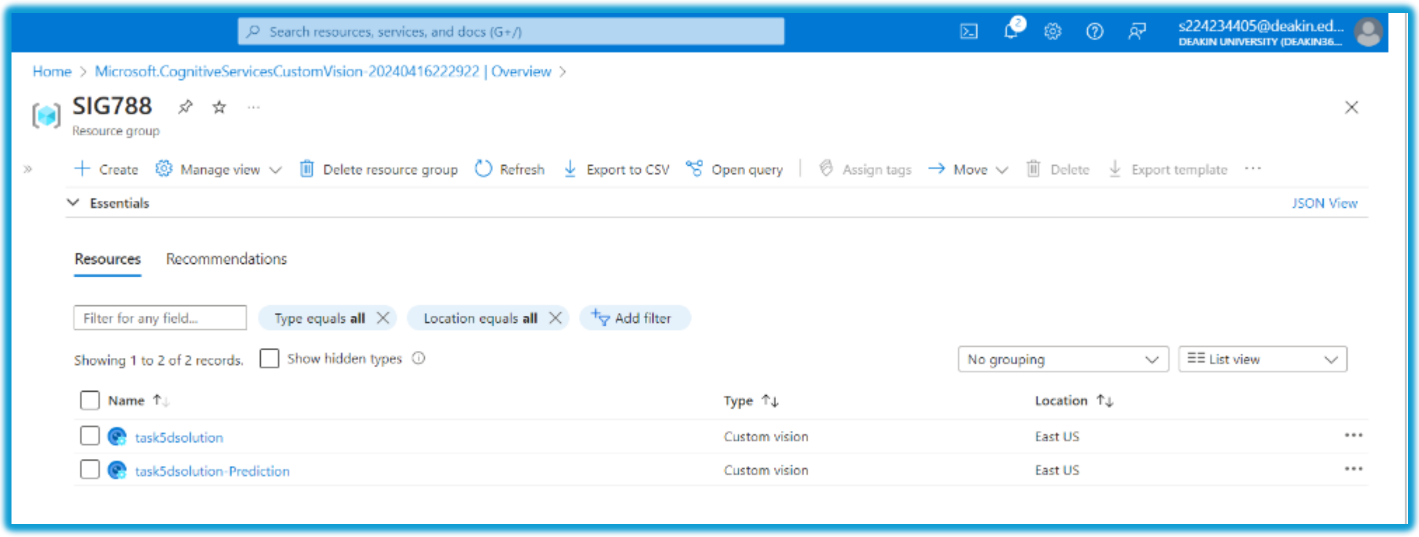
Importing necessary libraries

- Sets up the environment for working with the Azure Custom Vision service by importing required libraries and modules for integration, authentication, and data processing.
- Facilitates Azure Custom Vision integration by importing essential libraries and modules for authentication, data processing, and visualization.
- Prepares the code environment for Azure Custom Vision integration, including importing necessary libraries and modules for authentication, data processing, and visualization.



```python
# to connect to the training resource
from azure.cognitiveservices.vision.customvision.training import CustomVisionTrainingClient

# to connect to the prediction resource
from azure.cognitiveservices.vision.customvision.prediction import CustomVisionPredictionClient

# to send the input files in batch; and to identify the object regions; for the training of the model.
from azure.cognitiveservices.vision.customvision.training.models import ImageFileCreateBatch, ImageFileCreateEntry, Region

# To authenticate the client
from msrest.authentication import ApiKeyCredentials
import os, uuid

# To read the local environment variables and secret keys
import dotenv
from dotenv import load_dotenv, find_dotenv

# To read the dataset
import pandas as pd

# to view images
from IPython.display import Image as img

# to process data in batches
import itertools

# to draw bounding boxes in local output.
from PIL import Image, ImageDraw
import matplotlib.pyplot as plt
import numpy as np
import time
from matplotlib.patches import Rectangle

import IPython.display as ipd


# Function to send video frames to Custom Vision for vehicle detection
import cv2

from tqdm import tqdm
import subprocess
```

Create an Azure Custom Vision resource on the Azure Portal for both Training and Prediction



SIG788
Resource group

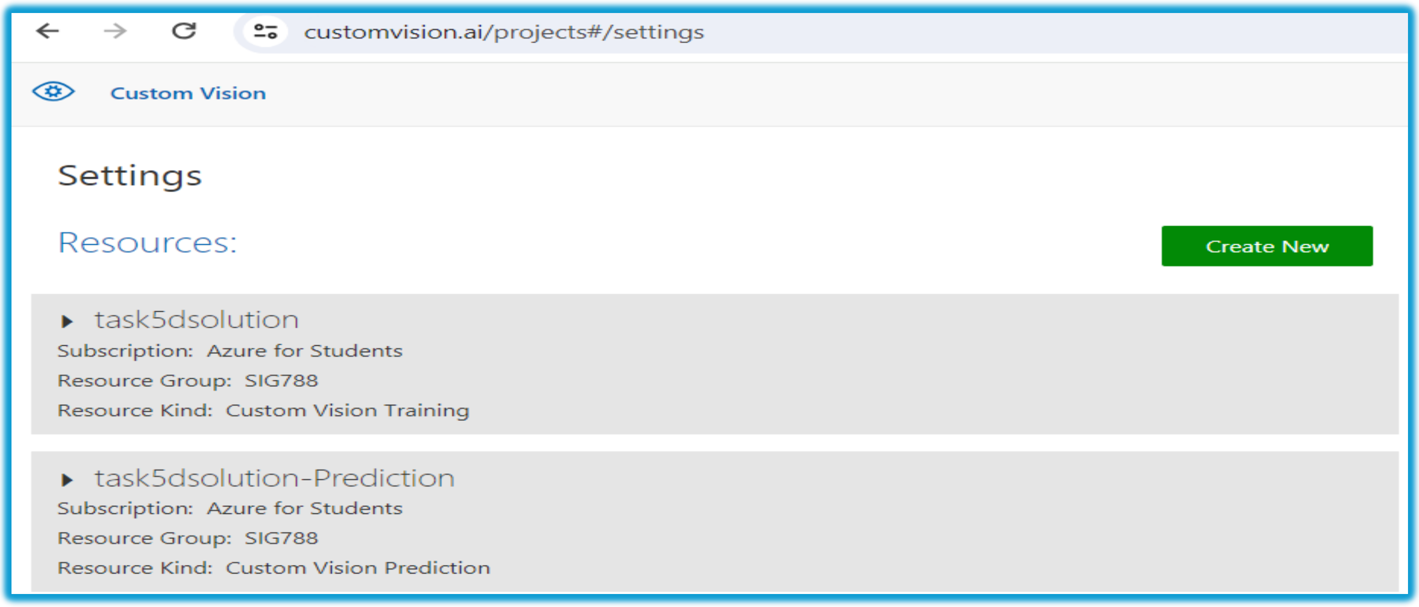
Resources | Recommendations

Showing 1 to 2 of 2 records. Show hidden types

Name	Type	Location
task5dsolution	Custom vision	East US
task5dsolution-Prediction	Custom vision	East US

Connect to the Custom Vision Service

Login to https://www.customvision.ai/projects#/settings and retrieve the below necessary variables. These variables hold the necessary endpoint URLs and authentication keys for accessing the Custom Vision training and prediction resources in Azure.

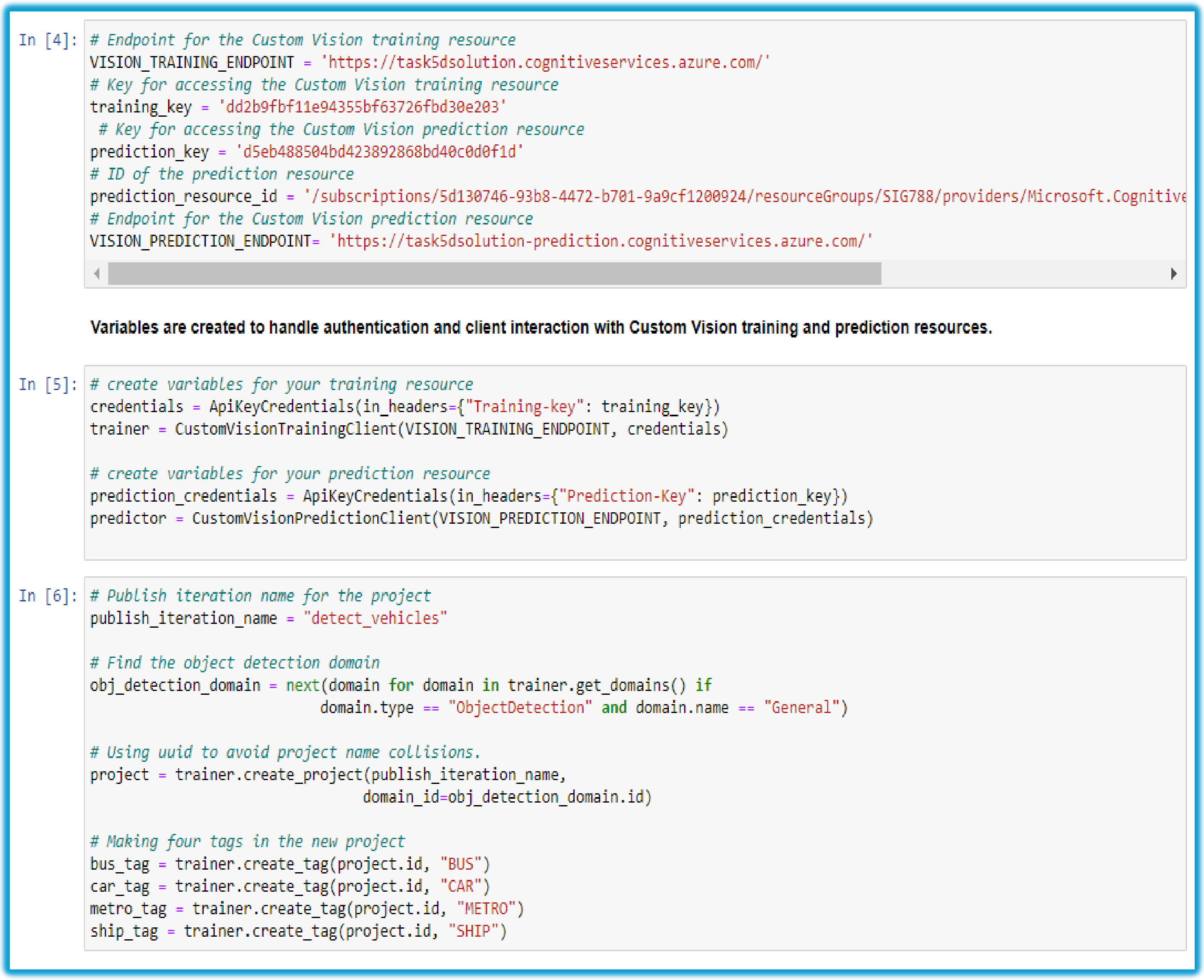


Settings

Resources:

task5dsolution
Subscription: Azure for Students
Resource Group: SIG788
Resource Kind: Custom Vision Training

task5dsolution-Prediction
Subscription: Azure for Students
Resource Group: SIG788
Resource Kind: Custom Vision Prediction



In [4]:
```
# Endpoint for the Custom Vision training resource
VISION_TRAINING_ENDPOINT = 'https://task5dsolution.cognitiveservices.azure.com/'
# Key for accessing the Custom Vision training resource
training_key = 'dd2b9fbf11e94355bf63726fbd30e203'
 # Key for accessing the Custom Vision prediction resource
prediction_key = 'd5eb488504bd423892868bd40c0d0f1d'
# ID of the prediction resource
prediction_resource_id = '/subscriptions/5d130746-93b8-4472-b701-9a9cf1200924/resourceGroups/SIG788/providers/Microsoft.Cognitive
# Endpoint for the Custom Vision prediction resource
VISION_PREDICTION_ENDPOINT= 'https://task5dsolution-prediction.cognitiveservices.azure.com/'
```

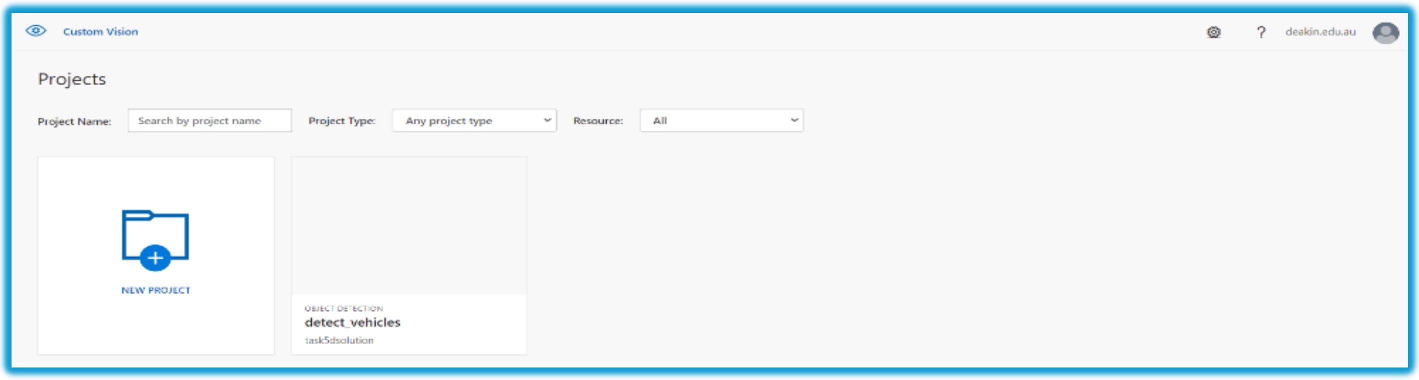
Variables are created to handle authentication and client interaction with Custom Vision training and prediction resources.

In [5]:
```
# create variables for your training resource
credentials = ApiKeyCredentials(in_headers={"Training-key": training_key})
trainer = CustomVisionTrainingClient(VISION_TRAINING_ENDPOINT, credentials)

# create variables for your prediction resource
prediction_credentials = ApiKeyCredentials(in_headers={"Prediction-Key": prediction_key})
predictor = CustomVisionPredictionClient(VISION_PREDICTION_ENDPOINT, prediction_credentials)
```

In [6]:
```
# Publish iteration name for the project
publish_iteration_name = "detect_vehicles"

# Find the object detection domain
obj_detection_domain = next(domain for domain in trainer.get_domains() if
                           domain.type == "ObjectDetection" and domain.name == "General")

# Using uuid to avoid project name collisions.
project = trainer.create_project(publish_iteration_name,
                                 domain_id=obj_detection_domain.id)

# Making four tags in the new project
bus_tag = trainer.create_tag(project.id, "BUS")
car_tag = trainer.create_tag(project.id, "CAR")
metro_tag = trainer.create_tag(project.id, "METRO")
ship_tag = trainer.create_tag(project.id, "SHIP")
```

View project

To view your project, go to https://www.customvision.ai/projects

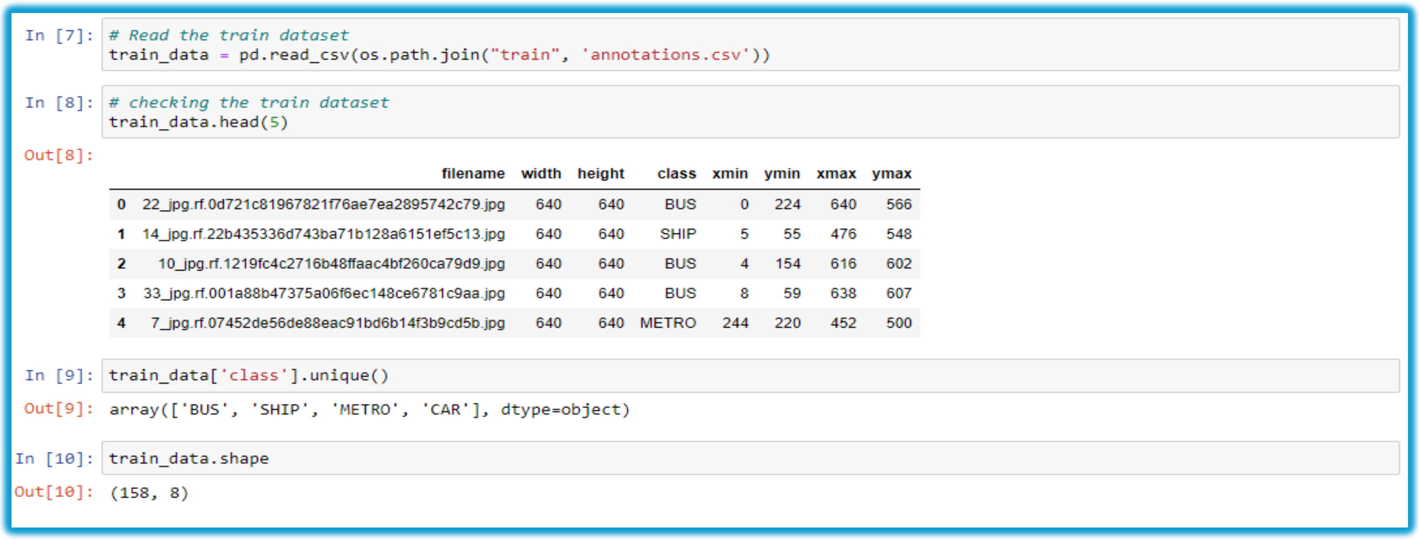


Populating the Project for Training

Once the project is created, it needs to be populated with images and their corresponding tags, along with bounding box coordinates if applicable. This data will be used to train the object detection model, enabling it to recognize and localize objects accurately within images.

Dataset preparation

The dataset annotations for the training data are read and parsed to extract object information like class labels and bounding box coordinates. This formatted data is then prepared for uploading to our project, ensuring it is compatible with the Custom Vision platform.



```
In [7]: # Read the train dataset
        train_data = pd.read_csv(os.path.join("train", 'annotations.csv'))

In [8]: # checking the train dataset
        train_data.head(5)
Out[8]:
```

	filename	width	height	class	xmin	ymin	xmax	ymax
0	22_jpg.rf.0d721c81967821f76ae7ea2895742c79.jpg	640	640	BUS	0	224	640	566
1	14_jpg.rf.22b435336d743ba71b128a6151ef5c13.jpg	640	640	SHIP	5	55	476	548
2	10_jpg.rf.1219fc4c2716b48ffaac4bf260ca79d9.jpg	640	640	BUS	4	154	616	602
3	33_jpg.rf.001a88b47375a06f6ec148ce6781c9aa.jpg	640	640	BUS	8	59	638	607
4	7_jpg.rf.07452de56de88eac91bd6b14f3b9cd5b.jpg	640	640	METRO	244	220	452	500

```
In [9]: train_data['class'].unique()
Out[9]: array(['BUS', 'SHIP', 'METRO', 'CAR'], dtype=object)

In [10]: train_data.shape
Out[10]: (158, 8)
```

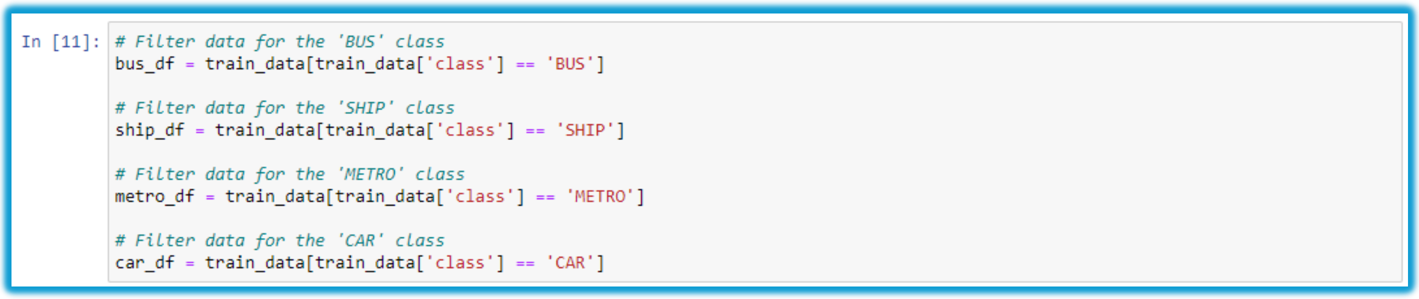
Dataset Observation

- There are four vehicle classification labels in the dataset - 'BUS', 'SHIP', 'METRO', 'CAR'
- The training dataset has 158 images.
- The annotations contain image width and height, the class it belongs to and the bounding box information.

Filtering Data by Class

The training data is filtered into separate data frames based on the class labels. This categorization allows for easier management and processing of data specific to each class:

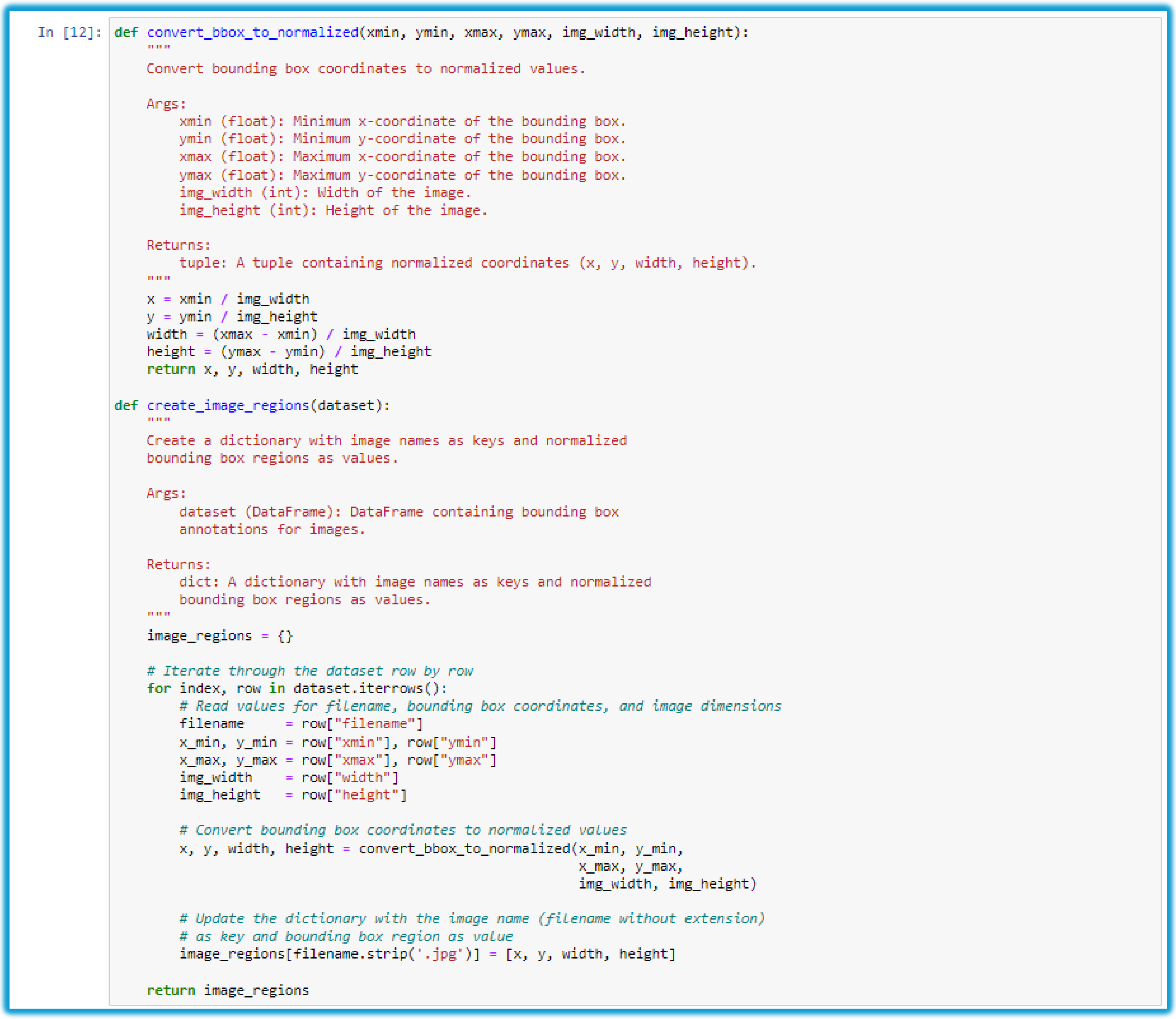
- bus_df: Contains data related to the 'BUS' class.
- ship_df: Contains data related to the 'SHIP' class.
- metro_df: Contains data related to the 'METRO' class.
- car_df: Contains data related to the 'CAR' class.



```
In [11]: # Filter data for the 'BUS' class
         bus_df = train_data[train_data['class'] == 'BUS']

         # Filter data for the 'SHIP' class
         ship_df = train_data[train_data['class'] == 'SHIP']

         # Filter data for the 'METRO' class
         metro_df = train_data[train_data['class'] == 'METRO']

         # Filter data for the 'CAR' class
         car_df = train_data[train_data['class'] == 'CAR']
```

Bounding Box Data Preparation

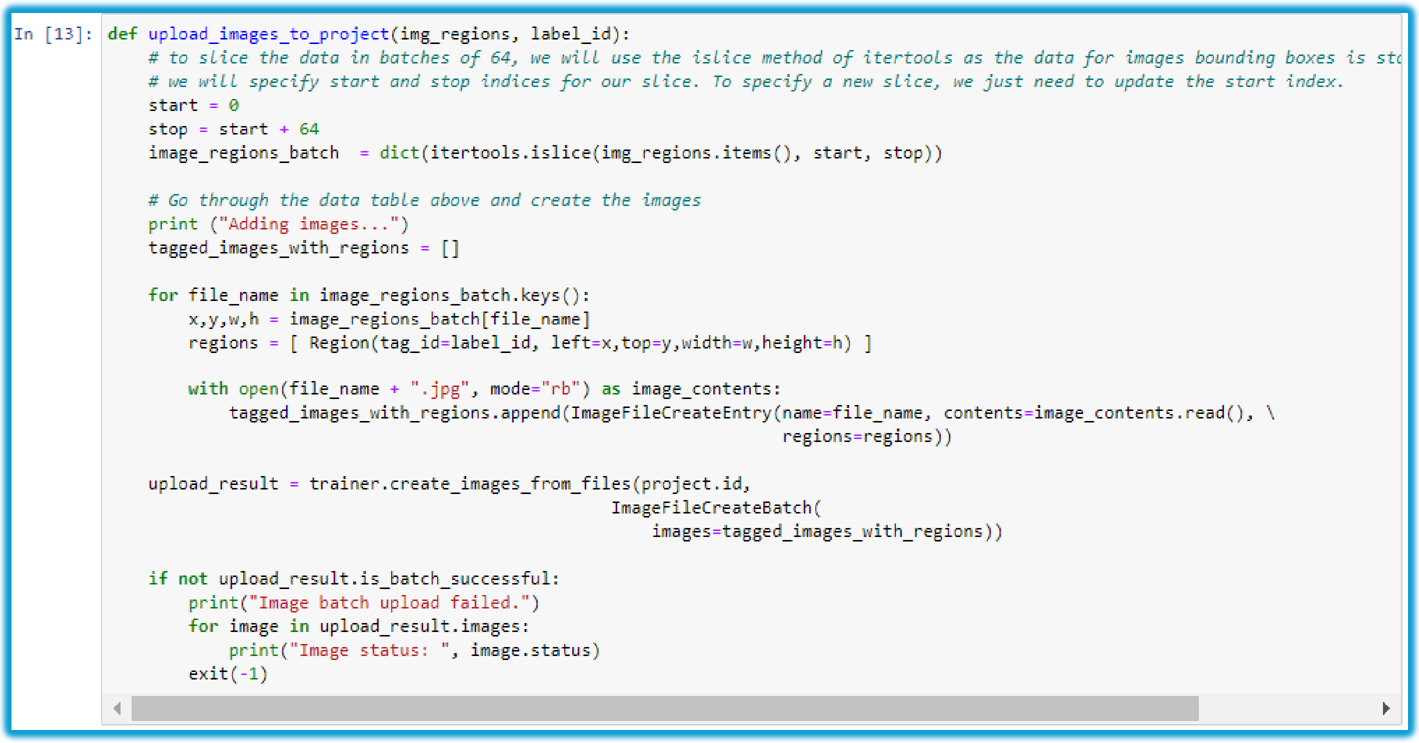
Functions are used to prepare bounding box data for training object detection models. They convert bounding box coordinates to a format suitable for training and create a dictionary representing the bounding box regions for each image in the dataset.



```
In [12]: def convert_bbox_to_normalized(xmin, ymin, xmax, ymax, img_width, img_height):
             """
             Convert bounding box coordinates to normalized values.

             Args:
                 xmin (float): Minimum x-coordinate of the bounding box.
                 ymin (float): Minimum y-coordinate of the bounding box.
                 xmax (float): Maximum x-coordinate of the bounding box.
                 ymax (float): Maximum y-coordinate of the bounding box.
                 img_width (int): Width of the image.
                 img_height (int): Height of the image.

             Returns:
                 tuple: A tuple containing normalized coordinates (x, y, width, height).
             """
             x = xmin / img_width
             y = ymin / img_height
             width = (xmax - xmin) / img_width
             height = (ymax - ymin) / img_height
             return x, y, width, height

         def create_image_regions(dataset):
             """
             Create a dictionary with image names as keys and normalized
             bounding box regions as values.

             Args:
                 dataset (DataFrame): DataFrame containing bounding box
                 annotations for images.

             Returns:
                 dict: A dictionary with image names as keys and normalized
                 bounding box regions as values.
             """
             image_regions = {}

             # Iterate through the dataset row by row
             for index, row in dataset.iterrows():
                 # Read values for filename, bounding box coordinates, and image dimensions
                 filename     = row["filename"]
                 x_min, y_min = row["xmin"], row["ymin"]
                 x_max, y_max = row["xmax"], row["ymax"]
                 img_width    = row["width"]
                 img_height   = row["height"]

                 # Convert bounding box coordinates to normalized values
                 x, y, width, height = convert_bbox_to_normalized(x_min, y_min,
                                                                  x_max, y_max,
                                                                  img_width, img_height)

                 # Update the dictionary with the image name (filename without extension)
                 # as key and bounding box region as value
                 image_regions[filename.strip('.jpg')] = [x, y, width, height]

             return image_regions
```

Upload images to the project

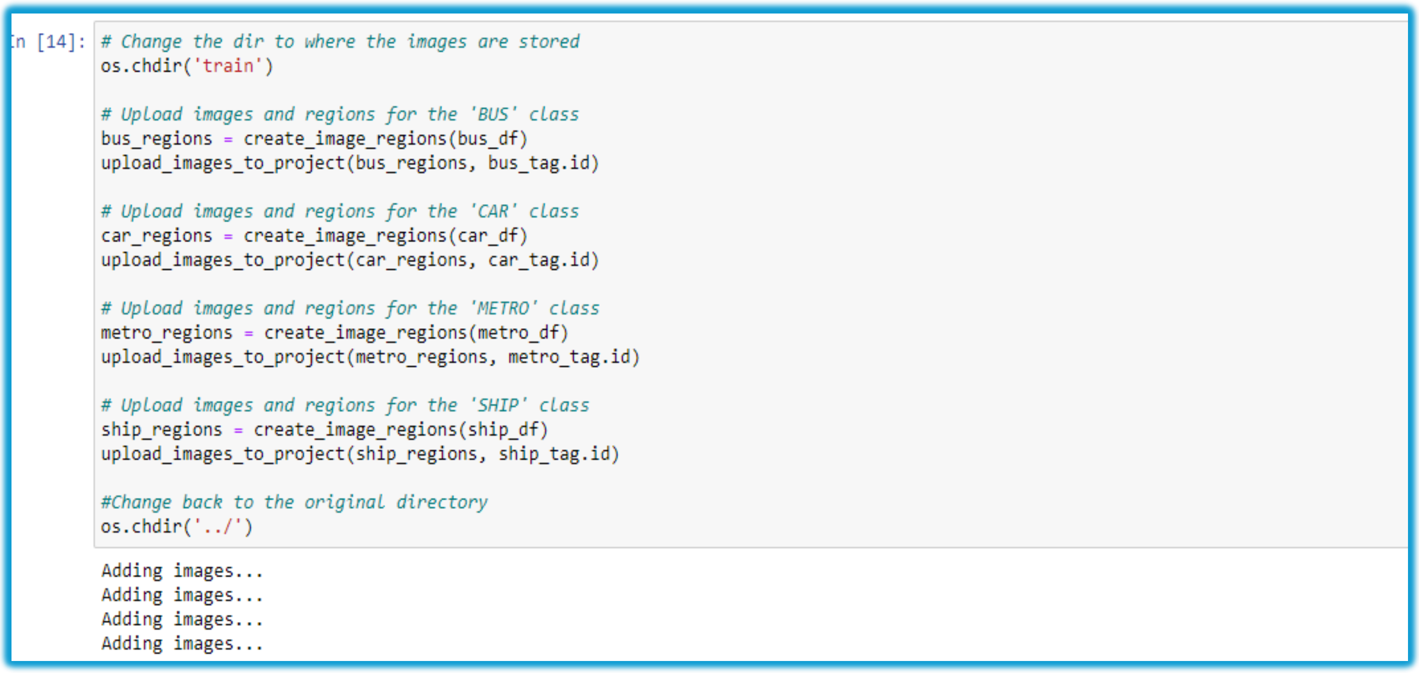
This function uploads images and their corresponding regions to the Custom Vision project. It slices the image regions into batches to avoid memory issues and uploads each batch of images and regions to the project. Error handling is included to handle failed batch uploads. As Azure places a limit of 64 entries at a time, we need to slice our train data into batches of 64 images and upload those.



```
In [13]: def upload_images_to_project(img_regions, label_id):
             # to slice the data in batches of 64, we will use the islice method of itertools as the data for images bounding boxes is st
             # we will specify start and stop indices for our slice. To specify a new slice, we just need to update the start index.
             start = 0
             stop = start + 64
             image_regions_batch  = dict(itertools.islice(img_regions.items(), start, stop))

             # Go through the data table above and create the images
             print ("Adding images...")
             tagged_images_with_regions = []

             for file_name in image_regions_batch.keys():
                 x,y,w,h = image_regions_batch[file_name]
                 regions = [ Region(tag_id=label_id, left=x,top=y,width=w,height=h) ]

                 with open(file_name + ".jpg", mode="rb") as image_contents:
                     tagged_images_with_regions.append(ImageFileCreateEntry(name=file_name, contents=image_contents.read(), \
                                                                            regions=regions))

             upload_result = trainer.create_images_from_files(project.id,
                                                               ImageFileCreateBatch(
                                                                   images=tagged_images_with_regions))

             if not upload_result.is_batch_successful:
                 print("Image batch upload failed.")
                 for image in upload_result.images:
                     print("Image status: ", image.status)
                 exit(-1)
```

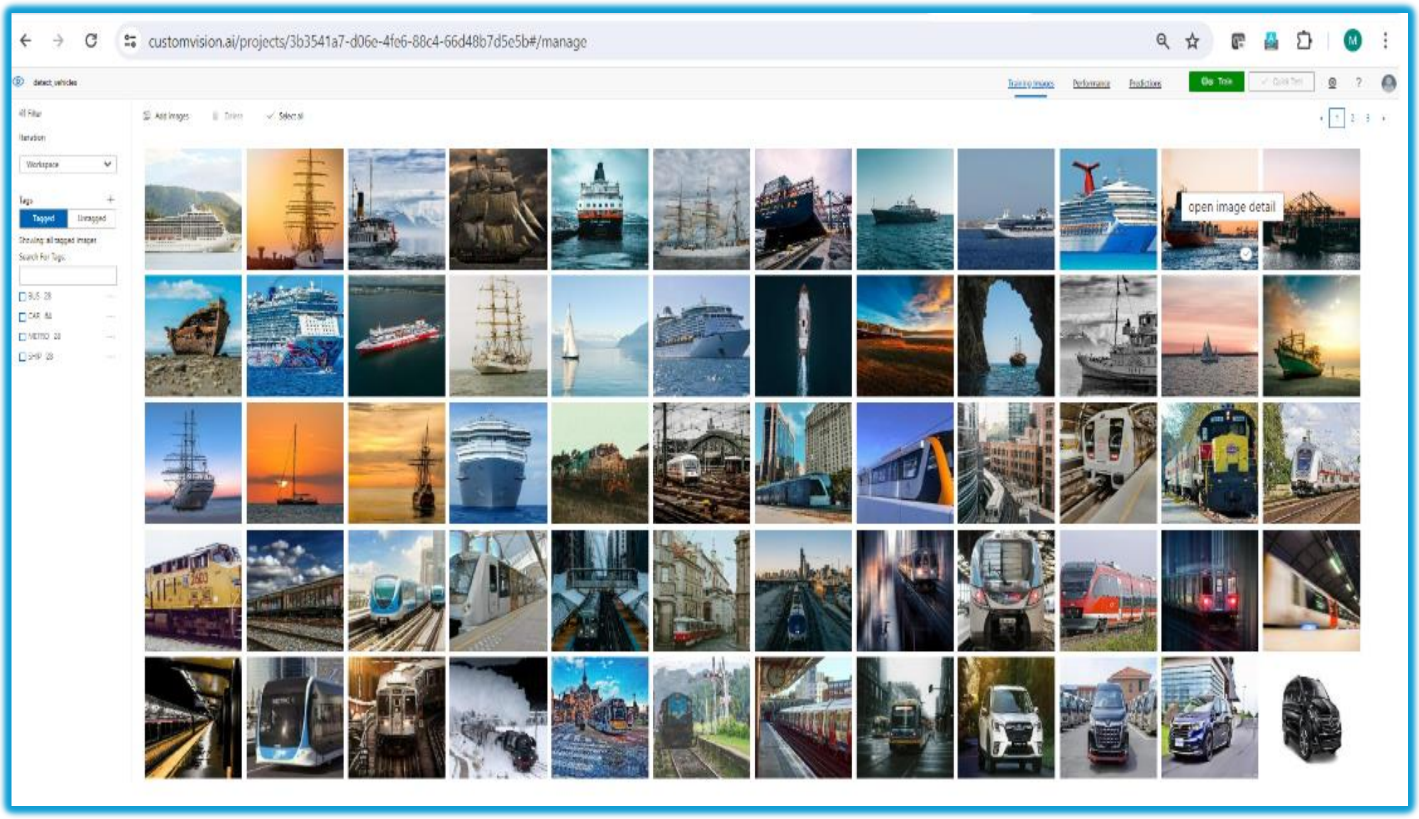
Upload Images for Different Classes

Upload images and their corresponding regions to the Custom Vision project for different classes ('BUS', 'CAR', 'METRO', 'SHIP'). First create image regions for each class using the create_image_region function and then upload these regions to the project using the upload_images_to_project function, specifying the tag ID for each class.

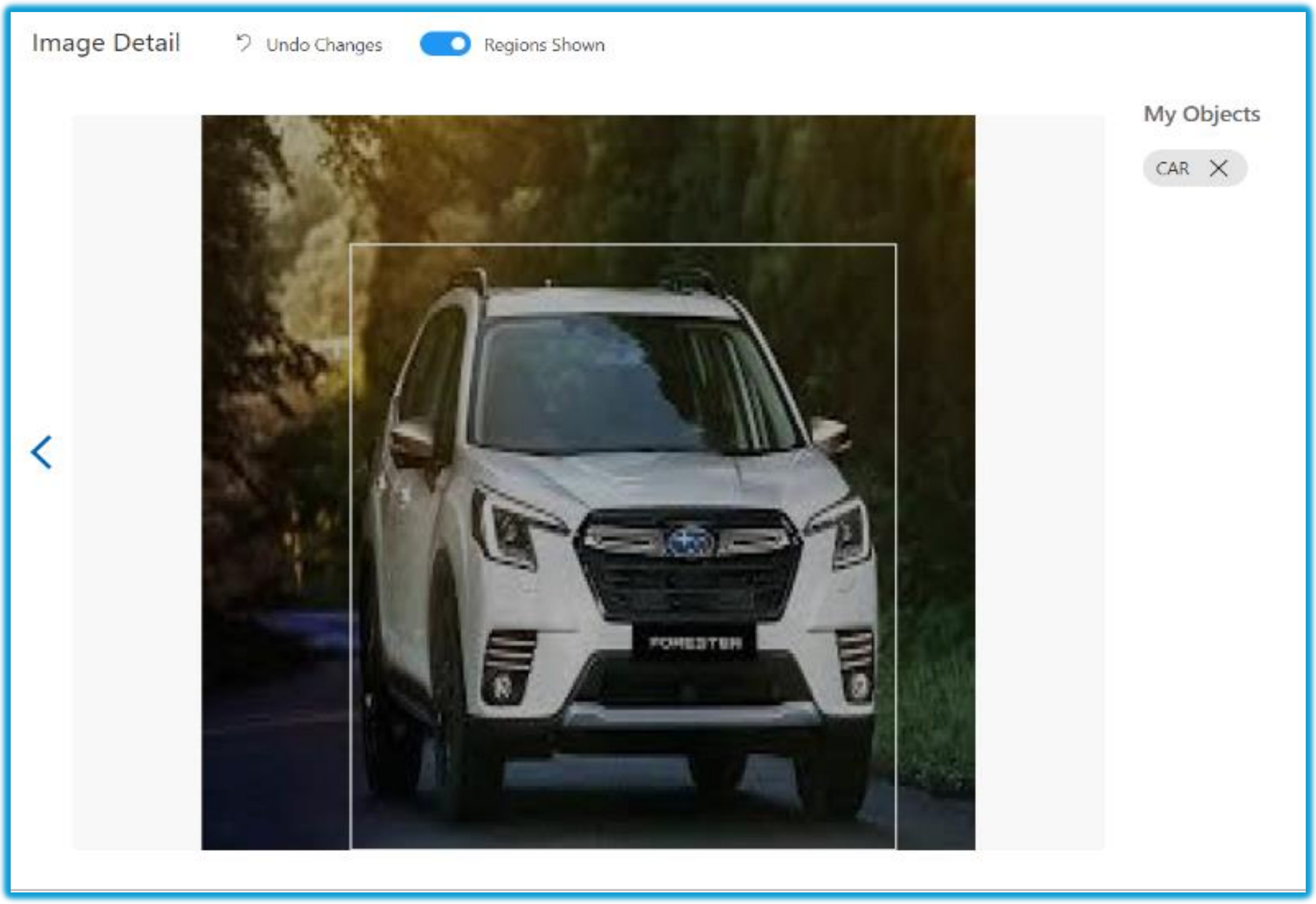


```
In [14]: # Change the dir to where the images are stored
         os.chdir('train')

         # Upload images and regions for the 'BUS' class
         bus_regions = create_image_regions(bus_df)
         upload_images_to_project(bus_regions, bus_tag.id)

         # Upload images and regions for the 'CAR' class
         car_regions = create_image_regions(car_df)
         upload_images_to_project(car_regions, car_tag.id)

         # Upload images and regions for the 'METRO' class
         metro_regions = create_image_regions(metro_df)
         upload_images_to_project(metro_regions, metro_tag.id)

         # Upload images and regions for the 'SHIP' class
         ship_regions = create_image_regions(ship_df)
         upload_images_to_project(ship_regions, ship_tag.id)

         #Change back to the original directory
         os.chdir('../')

         Adding images...
         Adding images...
         Adding images...
         Adding images...
```

Viewing Uploaded Images in Custom Vision Studio

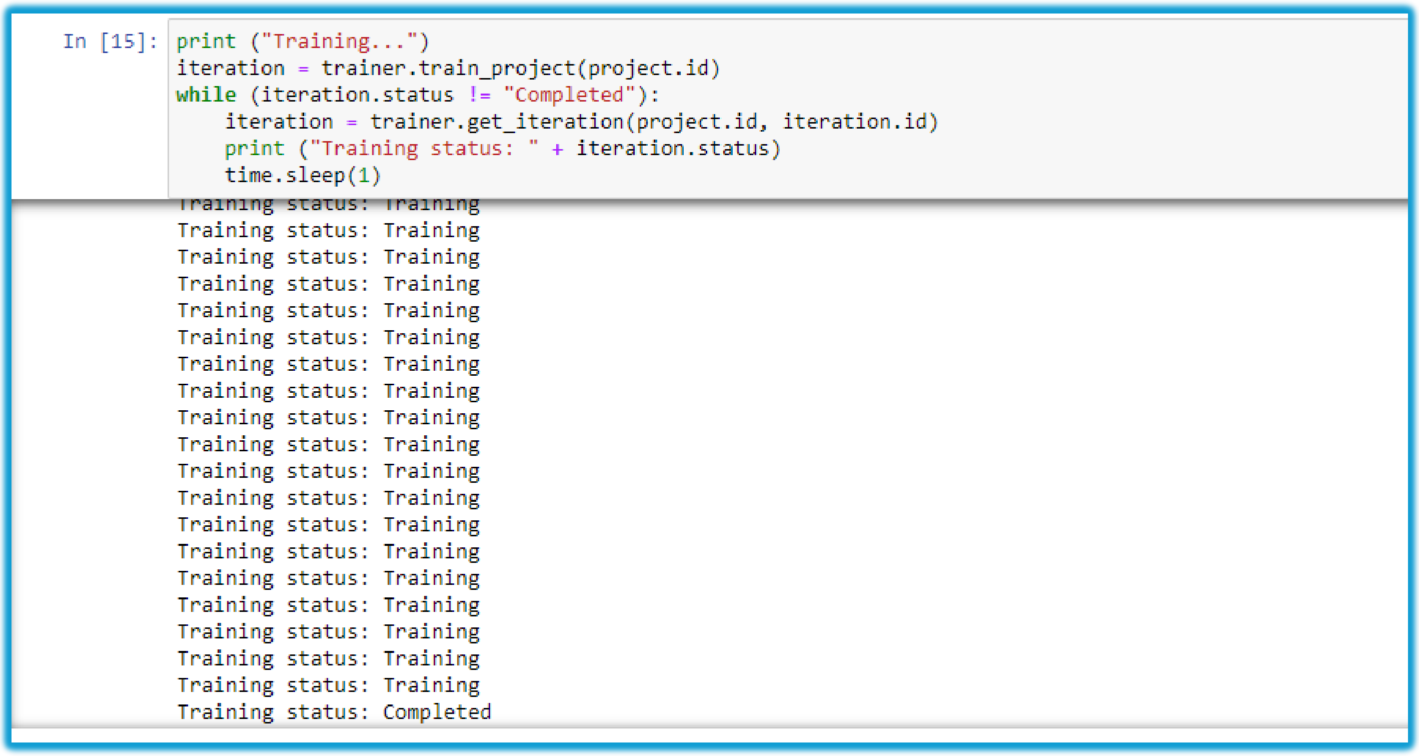


Click on individual images to view the bounding box and the label information



Training the model

Initiate the model training process for the project and continuously check the training status until it reaches completion. During each iteration, pause for 1 second between iterations to avoid excessive API requests.

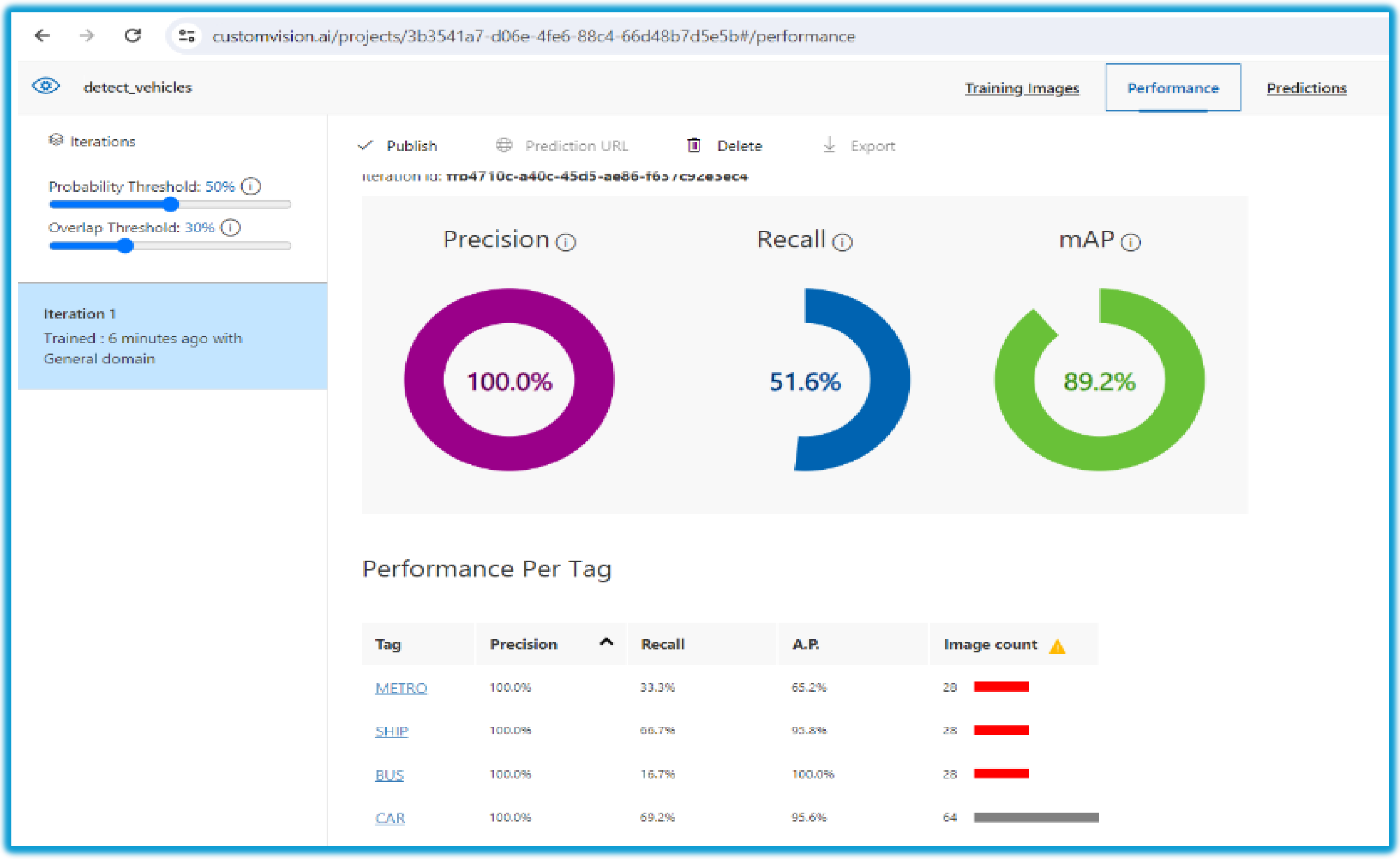


```
In [15]: print ("Training...")
         iteration = trainer.train_project(project.id)
         while (iteration.status != "Completed"):
             iteration = trainer.get_iteration(project.id, iteration.id)
             print ("Training status: " + iteration.status)
             time.sleep(1)
```

```
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Training
Training status: Completed
```

Viewing Model Performance in Custom Vision Studio

Upon completion of the training, the performance of the model can be assessed by navigating to the 'Performance' tab.



Probability Threshold: 50%

Overlap Threshold: 30%

Iteration 1
Trained : 6 minutes ago with General domain

Iteration id: ffb4710c-a40c-45d5-ae86-f657c92e3ec4

Precision 100.0%

Recall 51.6%

mAP 89.2%

Performance Per Tag

Tag	Precision	Recall	A.P.	Image count
METRO	100.0%	33.3%	65.2%	28
SHIP	100.0%	66.7%	93.8%	28
BUS	100.0%	16.7%	100.0%	28
CAR	100.0%	69.2%	95.6%	64

To observe the predictions, simply click on the tag name located under the 'Performance per tag' section. By selecting an image, predictions along with confidence scores and bounding boxes will be displayed.

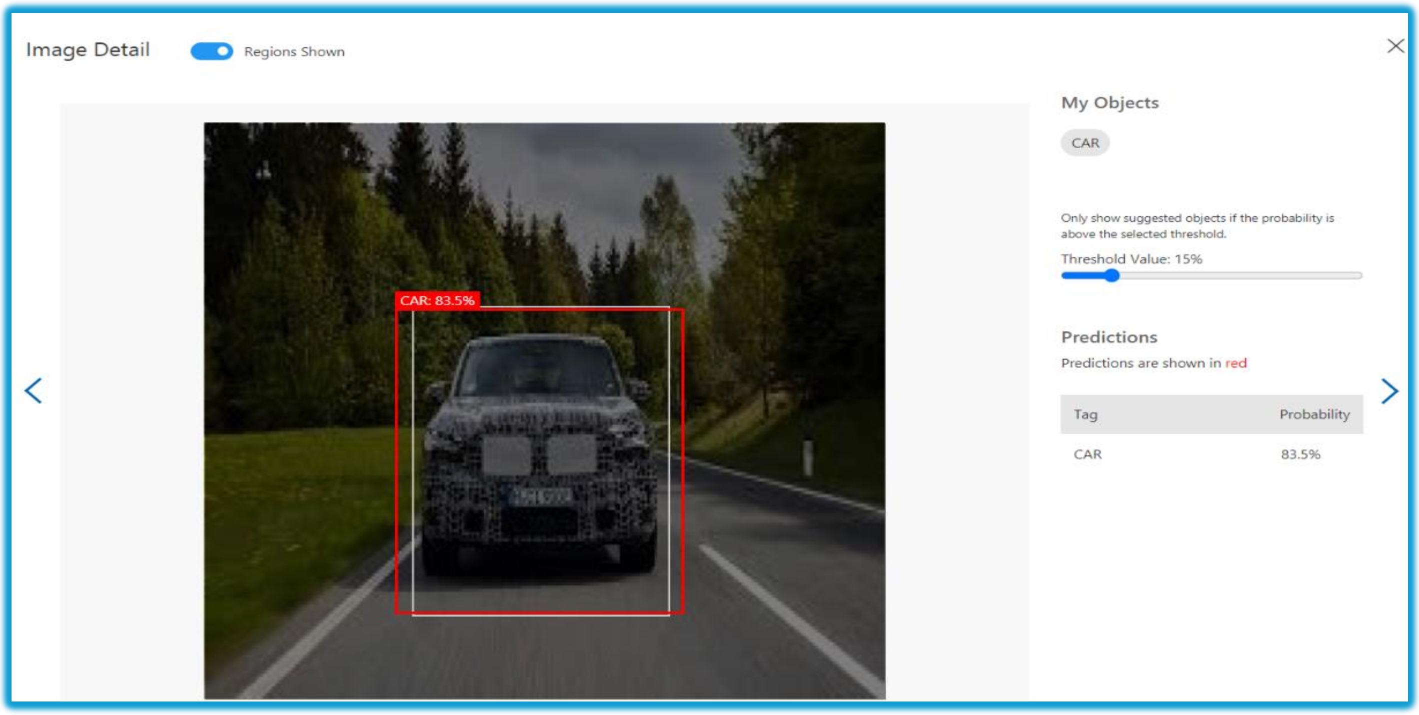


Image Detail
Regions Shown
CAR: 83.5%
My Objects
CAR
Only show suggested objects if the probability is above the selected threshold.
Threshold Value: 15%
Predictions
Predictions are shown in red

Tag	Probability
CAR	83.5%

Retraining the model involves taking a new slice of the training data and initiating the training process again. Once satisfied with the model's performance, publishing the iteration is necessary. In cases with multiple iterations, selecting the one with the best performance can be done by saving its iteration ID. In this scenario, since the current results are satisfactory, utilizing this iteration as the final one for publishing the model is appropriate.

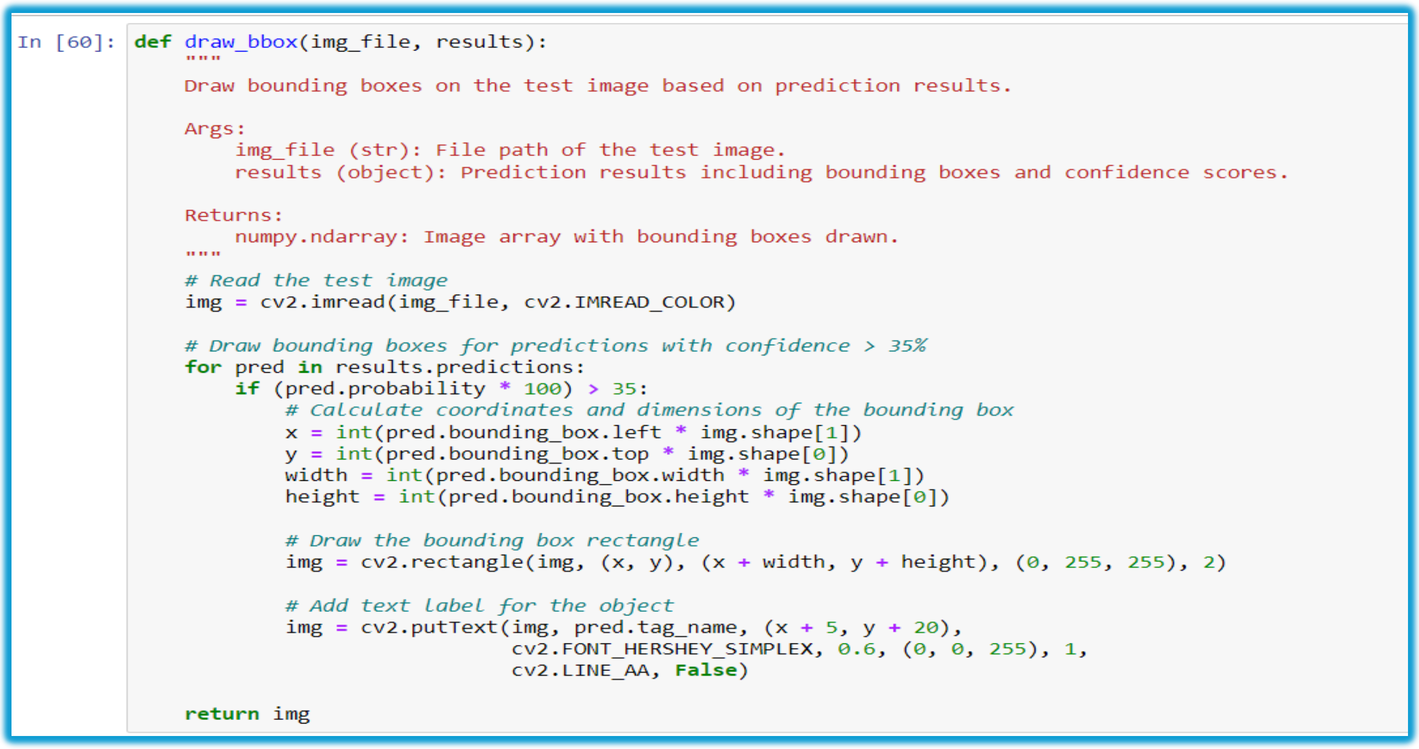
Publish the iteration

```
In [16]: # The iteration is now trained. Publish it to the project endpoint
         trainer.publish_iteration(project.id, iteration.id, publish_iteration_name, prediction_resource_id)
         print ("Done!")

         Done!
```

Object Detection Function: Make Predictions on Test Image

```
In [17]: def detect_image(img_file):
             """
             Detect objects in a test image using the trained model.

             Args:
                 img_file (str): File path of the test image.

             Returns:
                 dict: Prediction results including bounding boxes and confidence scores.
             """
             # Use the trained endpoint to make predictions
             # Open the sample image and retrieve prediction results
             with open(img_file, mode="rb") as test_data:
                 results = predictor.detect_image(project.id, publish_iteration_name, test_data)

             return results
```

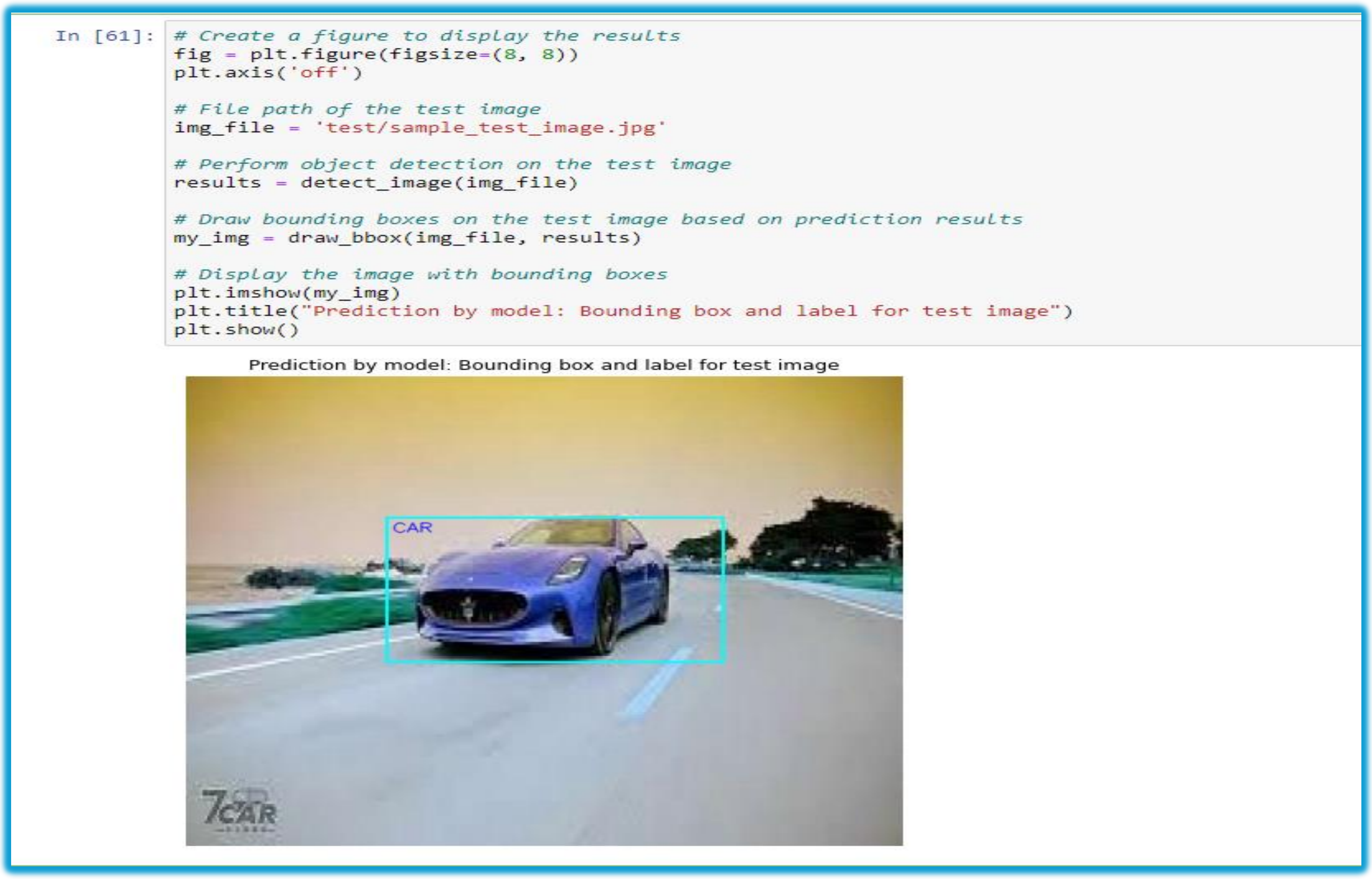
Draw bounding boxes on the test image based on prediction results



```
In [60]: def draw_bbox(img_file, results):
             """
             Draw bounding boxes on the test image based on prediction results.

             Args:
                 img_file (str): File path of the test image.
                 results (object): Prediction results including bounding boxes and confidence scores.

             Returns:
                 numpy.ndarray: Image array with bounding boxes drawn.
             """
             # Read the test image
             img = cv2.imread(img_file, cv2.IMREAD_COLOR)

             # Draw bounding boxes for predictions with confidence > 35%
             for pred in results.predictions:
                 if (pred.probability * 100) > 35:
                     # Calculate coordinates and dimensions of the bounding box
                     x = int(pred.bounding_box.left * img.shape[1])
                     y = int(pred.bounding_box.top * img.shape[0])
                     width = int(pred.bounding_box.width * img.shape[1])
                     height = int(pred.bounding_box.height * img.shape[0])

                     # Draw the bounding box rectangle
                     img = cv2.rectangle(img, (x, y), (x + width, y + height), (0, 255, 255), 2)

                     # Add text label for the object
                     img = cv2.putText(img, pred.tag_name, (x + 5, y + 20),
                                       cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 1,
                                       cv2.LINE_AA, False)

             return img
```

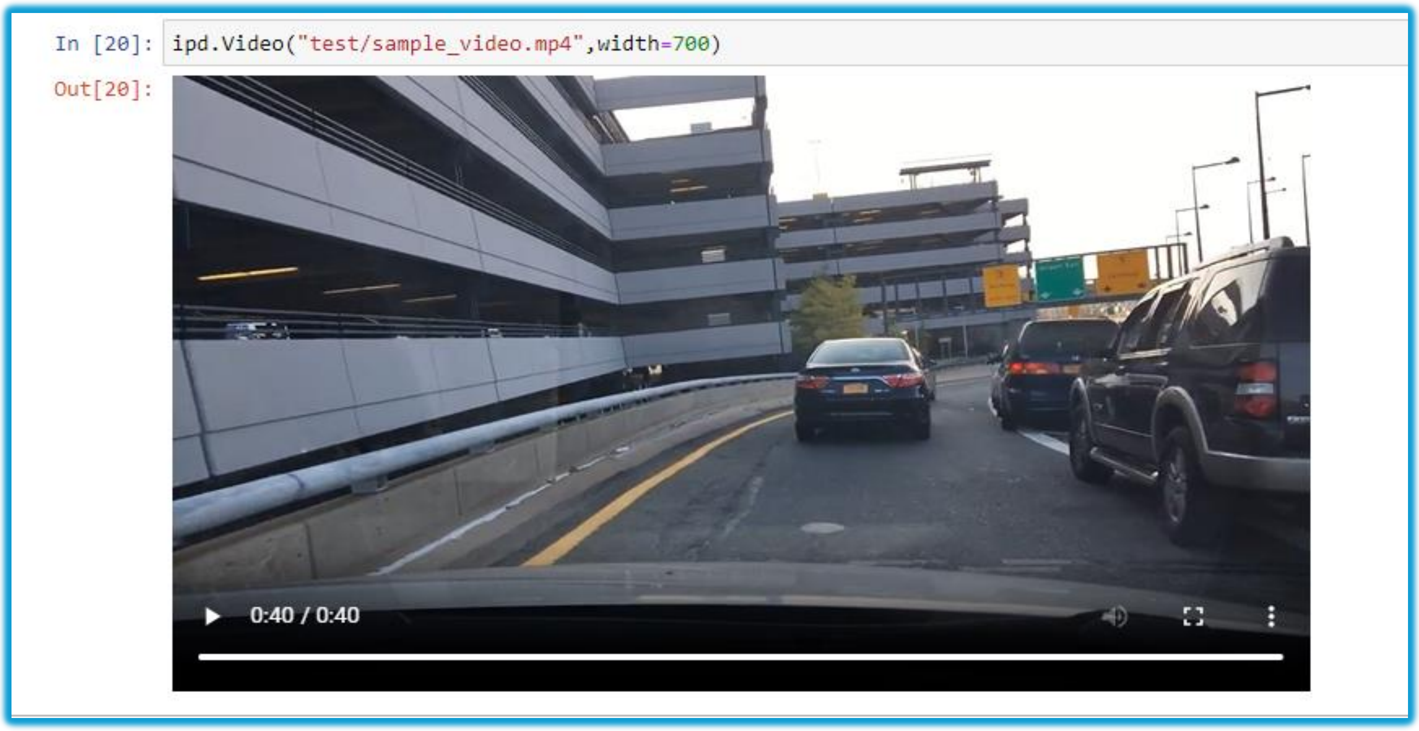
Test the model

To evaluate the model, you can generate predictions by sending a test image to the deployed endpoint. Among all the predictions made, only the bounding boxes with a confidence level exceeding a specified threshold, such as 35%, will be displayed.



```
In [61]: # Create a figure to display the results
         fig = plt.figure(figsize=(8, 8))
         plt.axis('off')

         # File path of the test image
         img_file = 'test/sample_test_image.jpg'

         # Perform object detection on the test image
         results = detect_image(img_file)

         # Draw bounding boxes on the test image based on prediction results
         my_img = draw_bbox(img_file, results)

         # Display the image with bounding boxes
         plt.imshow(my_img)
         plt.title("Prediction by model: Bounding box and label for test image")
         plt.show()
```

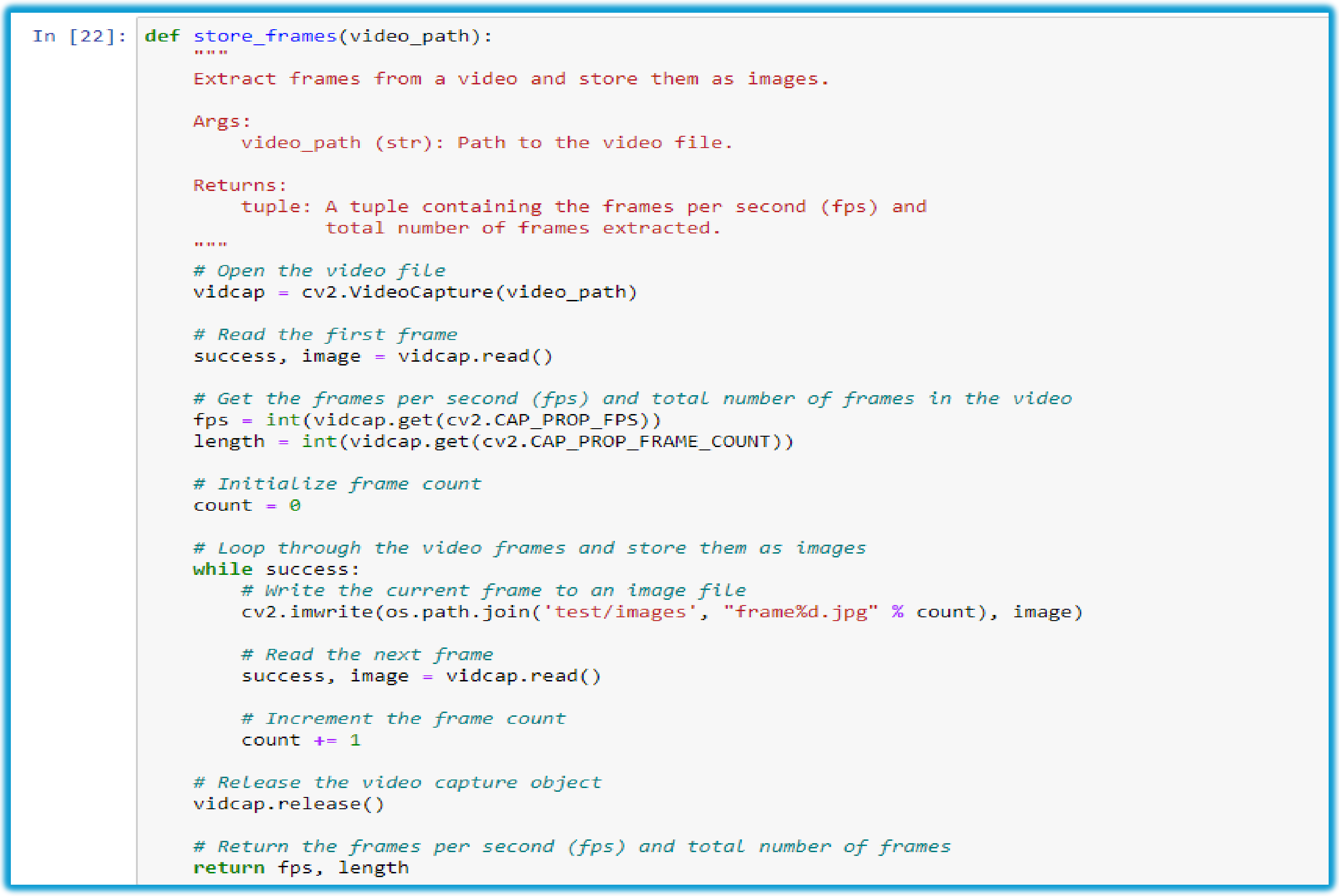
Display the sample video



```
In [20]: ipd.Video("test/sample_video.mp4",width=700)
Out[20]:
```

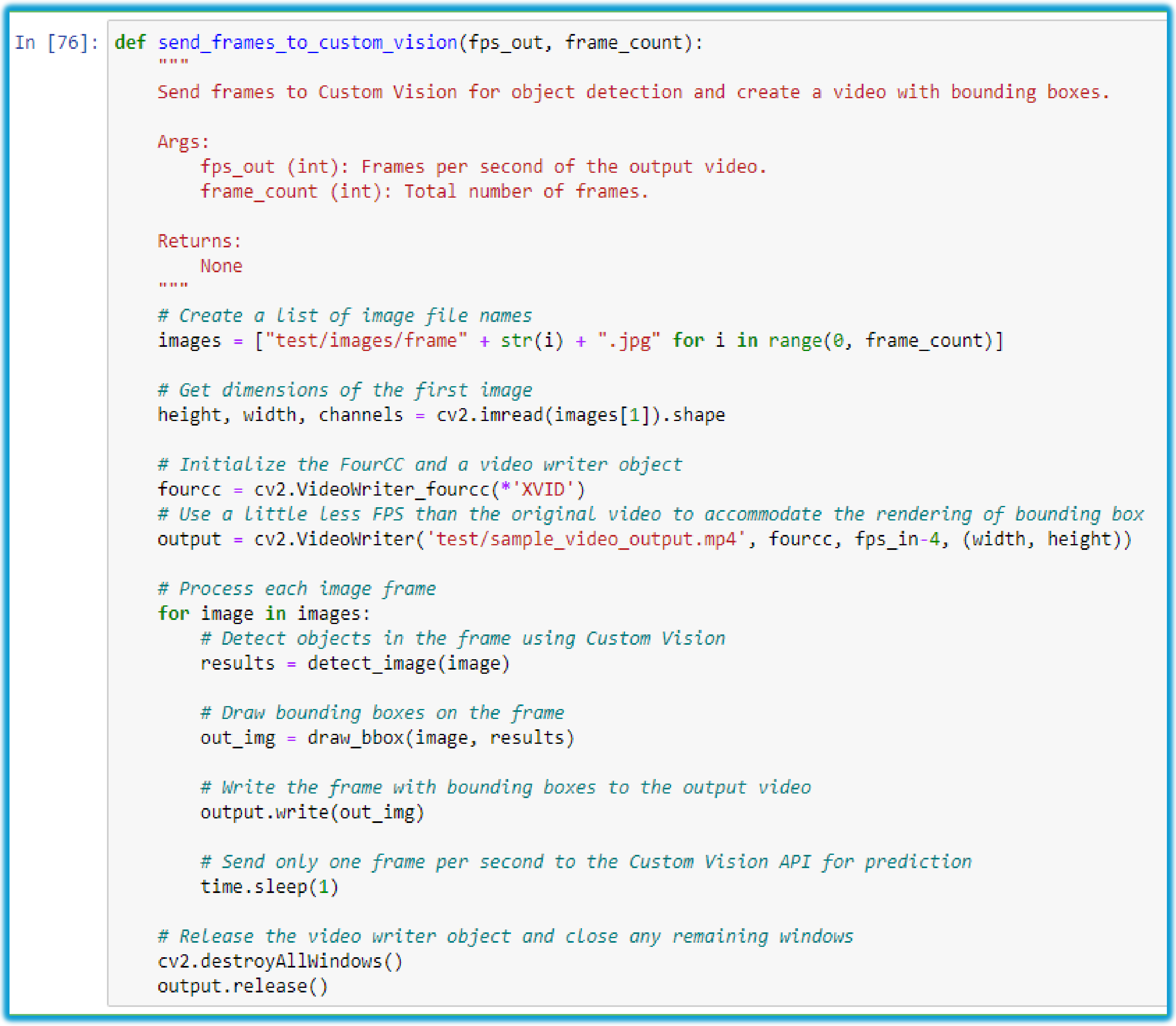
Extract frames from a video and store them as images

This function store_frames extracts frames from a video file and stores them as images. Returns a tuple containing the frames per second (fps) and the total number of frames extracted. This is needed to be used while creating the output video with bbox.



```
In [22]: def store_frames(video_path):
             """
             Extract frames from a video and store them as images.

             Args:
                 video_path (str): Path to the video file.

             Returns:
                 tuple: A tuple containing the frames per second (fps) and
                        total number of frames extracted.
             """
             # Open the video file
             vidcap = cv2.VideoCapture(video_path)

             # Read the first frame
             success, image = vidcap.read()

             # Get the frames per second (fps) and total number of frames in the video
             fps = int(vidcap.get(cv2.CAP_PROP_FPS))
             length = int(vidcap.get(cv2.CAP_PROP_FRAME_COUNT))

             # Initialize frame count
             count = 0

             # Loop through the video frames and store them as images
             while success:
                 # Write the current frame to an image file
                 cv2.imwrite(os.path.join('test/images', "frame%d.jpg" % count), image)

                 # Read the next frame
                 success, image = vidcap.read()

                 # Increment the frame count
                 count += 1

             # Release the video capture object
             vidcap.release()

             # Return the frames per second (fps) and total number of frames
             return fps, length
```

Send frames to Custom Vision for object detection and create a video with bounding boxes



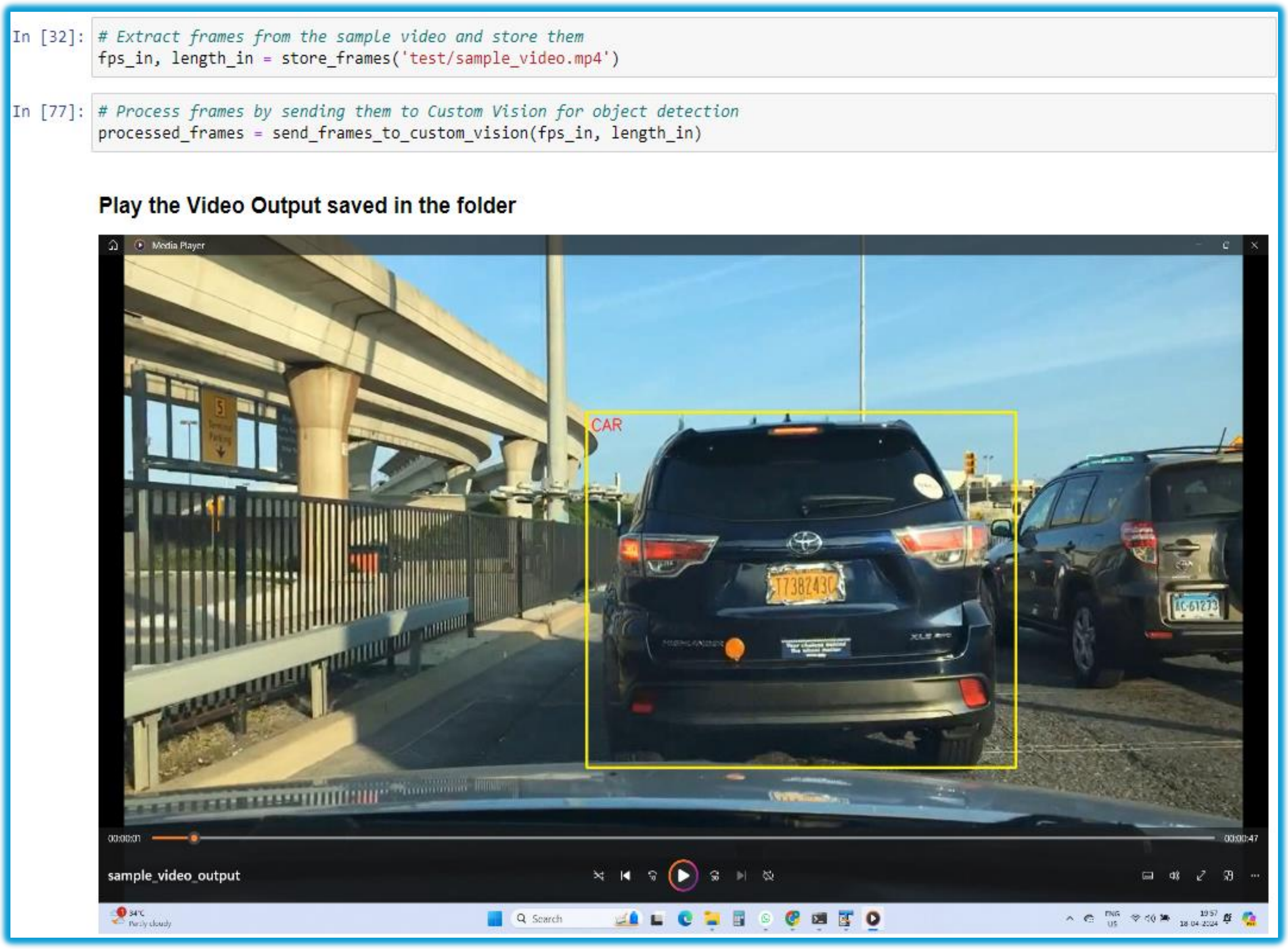
```
In [76]: def send_frames_to_custom_vision(fps_out, frame_count):
             """
             Send frames to Custom Vision for object detection and create a video with bounding boxes.

             Args:
                 fps_out (int): Frames per second of the output video.
                 frame_count (int): Total number of frames.

             Returns:
                 None
             """
             # Create a list of image file names
             images = ["test/images/frame" + str(i) + ".jpg" for i in range(0, frame_count)]

             # Get dimensions of the first image
             height, width, channels = cv2.imread(images[1]).shape

             # Initialize the FourCC and a video writer object
             fourcc = cv2.VideoWriter_fourcc(*'XVID')
             # Use a little less FPS than the original video to accommodate the rendering of bounding box
             output = cv2.VideoWriter('test/sample_video_output.mp4', fourcc, fps_in-4, (width, height))

             # Process each image frame
             for image in images:
                 # Detect objects in the frame using Custom Vision
                 results = detect_image(image)

                 # Draw bounding boxes on the frame
                 out_img = draw_bbox(image, results)

                 # Write the frame with bounding boxes to the output video
                 output.write(out_img)

                 # Send only one frame per second to the Custom Vision API for prediction
                 time.sleep(1)

             # Release the video writer object and close any remaining windows
             cv2.destroyAllWindows()
             output.release()
```

Detect objects in the sample video

Since the requirement is to send only one frame per second to the Custom Vision API, there are two options:

- Skipping Frames: If the FPS of the input video is around 29, you would need to drop 28 frames and process only one frame per second. However, this would result in a significant reduction in the number of frames in the output video, potentially affecting the quality of the object detection.
- Extracting and Processing All Frames: Another option is to extract all frames from the sample video and store them in a list. Then, loop through all the images every second and create the output video. This approach ensures that all frames are processed, but it may lead to increased processing time and resource utilization.

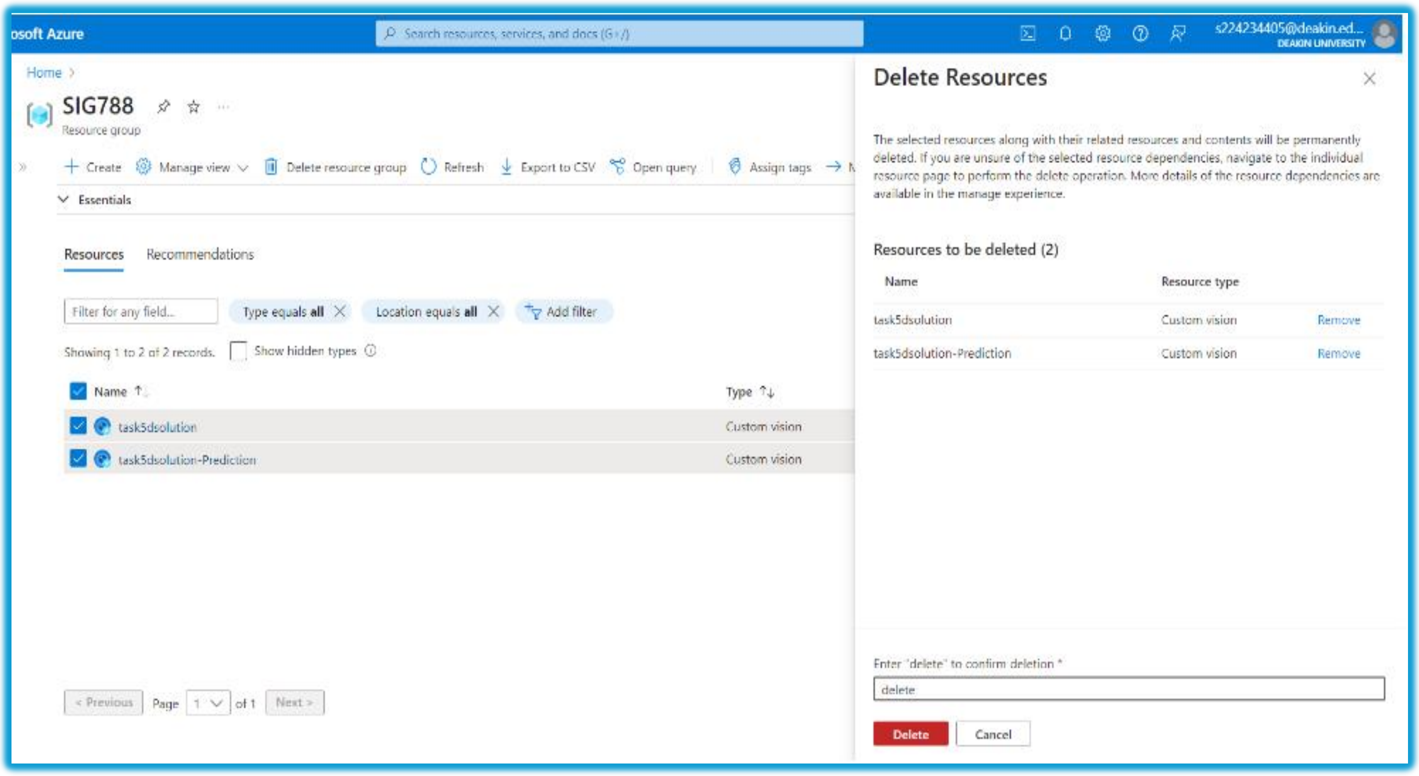
You would need to consider the trade-offs between processing efficiency and output video quality when deciding which approach to use. For our task, since the video is of small duration, we will choose second approach.



```
In [32]: # Extract frames from the sample video and store them
         fps_in, length_in = store_frames('test/sample_video.mp4')

In [77]: # Process frames by sending them to Custom Vision for object detection
         processed_frames = send_frames_to_custom_vision(fps_in, length_in)
```

Play the Video Output saved in the folder

Close the project - resource deletion



SIG788
Resource group

Showing 1 to 2 of 2 records. Show hidden types

Name	Type
✓ task5dsolution	Custom vision
✓ task5dsolution-Prediction	Custom vision

Delete Resources

The selected resources along with their related resources and contents will be permanently deleted. If you are unsure of the selected resource dependencies, navigate to the individual resource page to perform the delete operation. More details of the resource dependencies are available in the manage experience.

Resources to be deleted (2)

Name	Resource type	
task5dsolution	Custom vision	Remove
task5dsolution-Prediction	Custom vision	Remove

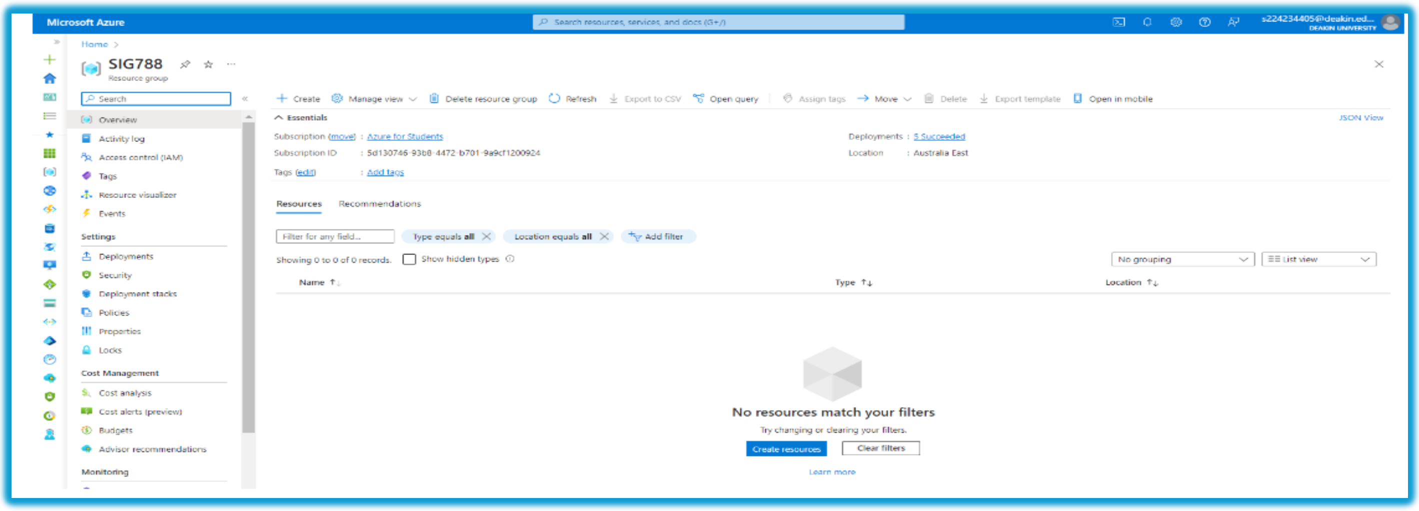
Enter 'delete' to confirm deletion *

delete

Delete Cancel

Resource Deletion Confirmation

Verified that all resources have been successfully deleted.



SIG788
Resource group

Essentials

Subscription (move) : Azure for Students
Subscription ID : 5d130746-93b8-4472-b701-9a9cf1200924
Tags (edit) : Add tags
Deployments : 5 Succeeded
Location : Australia East

Showing 0 to 0 of 0 records.

No resources match your filters
Try changing or clearing your filters.

Improvement Areas

To improve the performance of the model and enable it to detect cars from different angles in videos, consider the following improvement areas:

- Data Augmentation: Generate additional training data by applying transformations such as rotation, scaling, translation, flipping, and blurring to the existing vehicle images. This will help the model generalize better to different angles and conditions.
- Diverse Training Data: Collect a more diverse dataset that includes vehicle images captured from various angles, distances, lighting conditions, and backgrounds. This will expose the model to a wider range of scenarios and improve its ability to recognize vehicles in different situations.
- Annotated Bounding Boxes: Ensure accurate and precise annotation of bounding boxes around vehicles in the training images. Correctly annotated data is essential for training an object detection model effectively.
- Data Filtering and Cleaning: Remove noisy or irrelevant data from the training dataset to prevent the model from learning from irrelevant features. Focus on high-quality, relevant images that represent the target concept well.
- Data Balancing: Ensure that the dataset is balanced across different classes (e.g., car, bus, truck) to prevent the model from being biased towards the majority class. Use techniques such as oversampling, under sampling, or class weighting to address class imbalance.
- Cross-Validation: Implement cross-validation techniques to evaluate the model's performance more accurately and ensure robustness across different subsets of the data.
- Regularization Techniques: Apply regularization techniques such as dropout, L1/L2 regularization, or data augmentation-based regularization to prevent overfitting and improve generalization performance.

References

1. Kaggle. (n.d.). Driving Video Object Tracking. Retrieved from https://www.kaggle.com/code/kirollosashraf/driving-video-object-tracking
2. Microsoft. (n.d.). Quickstart: Use the Custom Vision client library for Python. Retrieved from https://learn.microsoft.com/en-us/azure/ai-services/custom-vision-service/quickstarts/image-classification?tabs=windows%2Cvisual-studio&pivots=programming-language-python