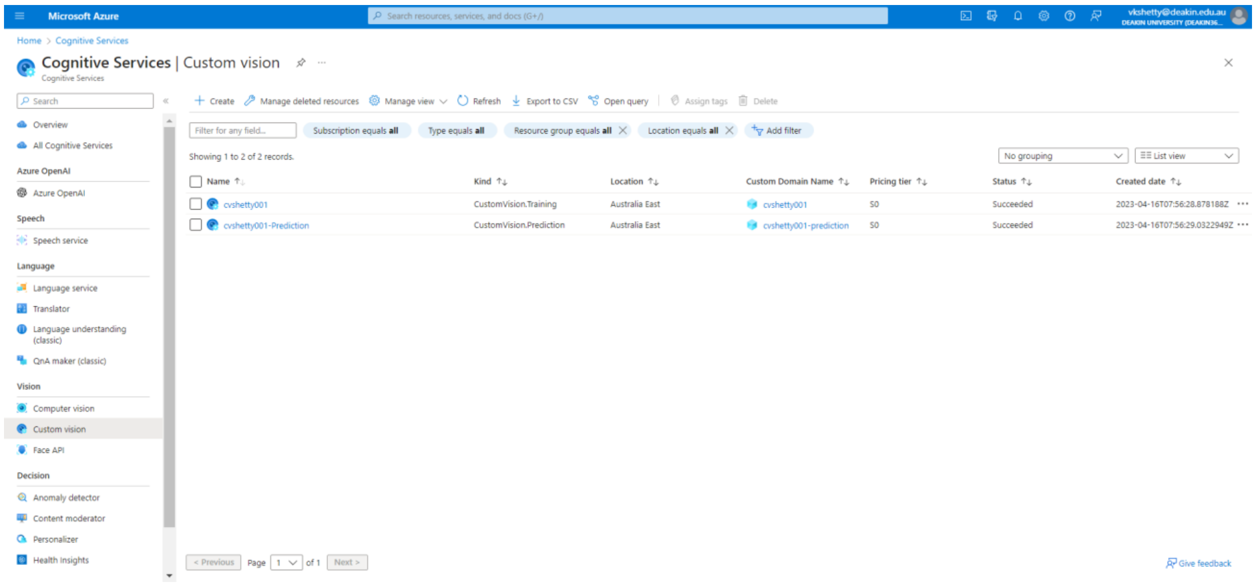
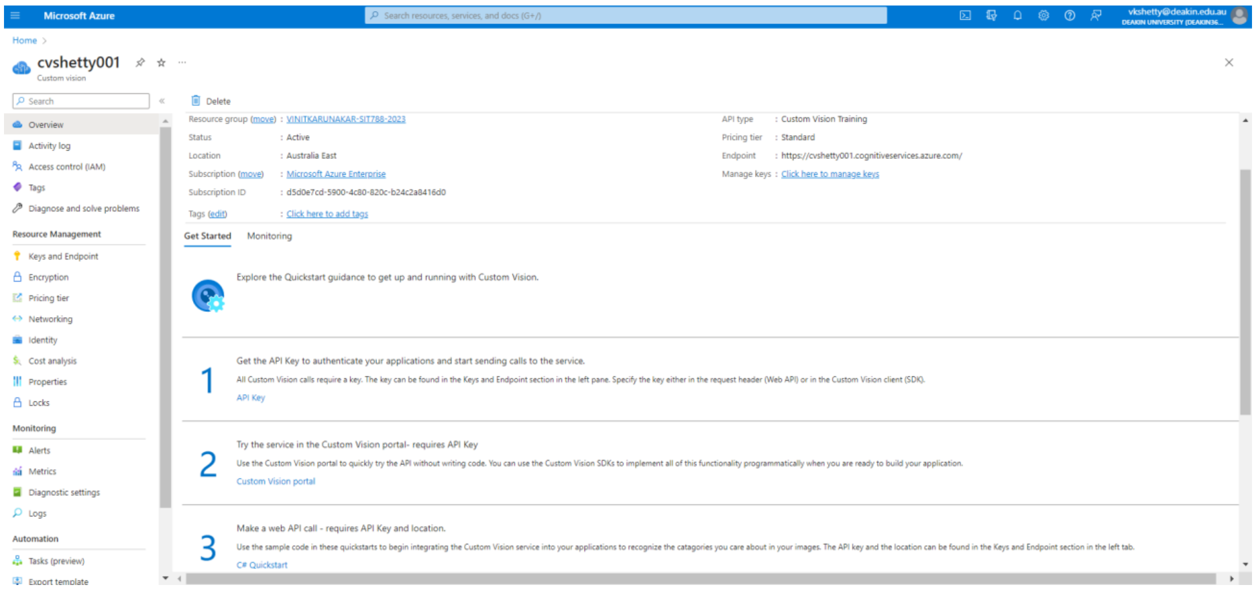
Computer vision and custom vision

1. Creating a Custom Vision resource



Showing 1 to 2 of 2 records.

Name	Kind	Location	Custom Domain Name	Pricing tier	Status	Created date
cvshetty001	CustomVision.Training	Australia East	cvshetty001	S0	Succeeded	2023-04-16T07:56:28.878188Z
cvshetty001-Prediction	CustomVision.Prediction	Australia East	cvshetty001-prediction	S0	Succeeded	2023-04-16T07:56:29.0322949Z



Resource group (move) : VINITKARUNAKAR-SIT788-2023
Status : Active
Location : Australia East
Subscription (move) : Microsoft Azure Enterprise
Subscription ID : d5d0e7cd-5900-4c80-820c-b24c2a8416d0
Tags (edit) : Click here to add tags

API type : Custom Vision Training
Pricing tier : Standard
Endpoint : https://cvshetty001.cognitiveservices.azure.com/
Manage keys : Click here to manage keys

Explore the Quickstart guidance to get up and running with Custom Vision.

1 Get the API Key to authenticate your applications and start sending calls to the service.
All Custom Vision calls require a key. The key can be found in the Keys and Endpoint section in the left pane. Specify the key either in the request header (Web API) or in the Custom Vision client (SDK).
API Key

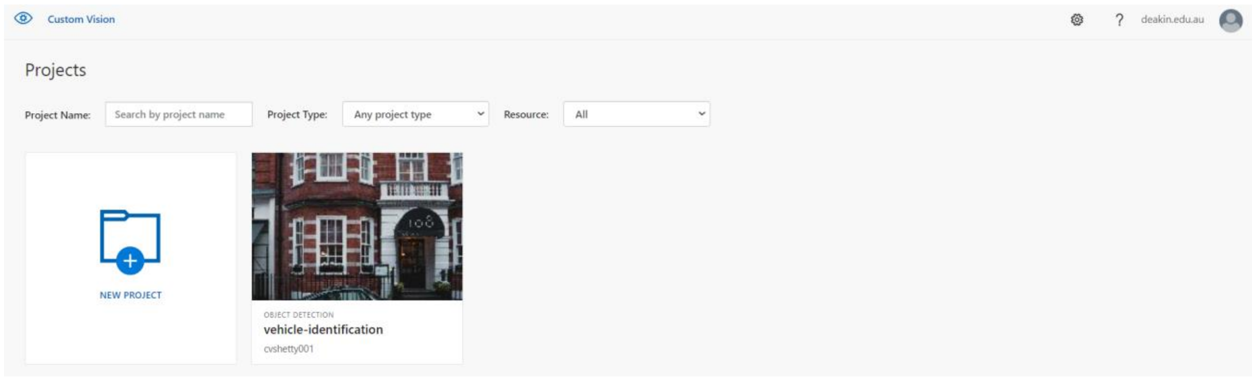
2 Try the service in the Custom Vision portal- requires API Key
Use the Custom Vision portal to quickly try the API without writing code. You can use the Custom Vision SDKs to implement all of this functionality programmatically when you are ready to build your application.
Custom Vision portal

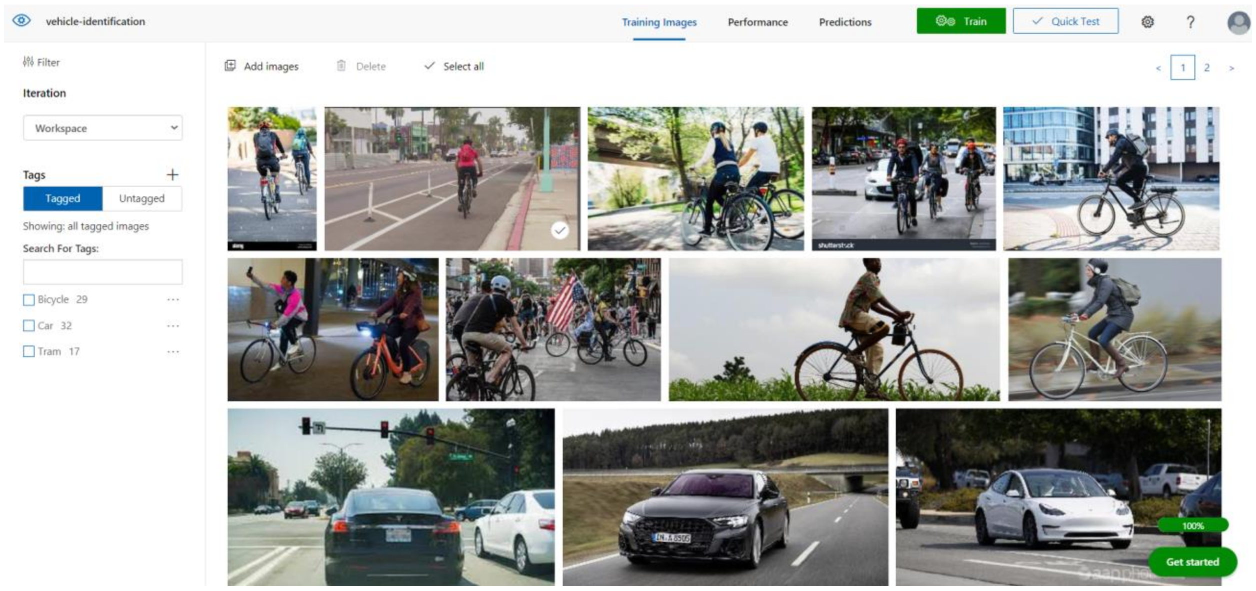
3 Make a web API call - requires API Key and location.
Use the sample code in these quickstarts to begin integrating the Custom Vision service into your applications to recognize the catagories you care about in your images. The API key and the location can be found in the Keys and Endpoint section in the left tab.
C# Quickstart

We have created a custom vision resource in Azure “**cvshetty001**” which can be used to train and deploy machine learning models for image classification, object detection, and other computer vision tasks.

To begin with, training the model, we click on the Custom Vision Portal where we start building a custom vision project and further training the model.

2. Creating the project and training the data with images.

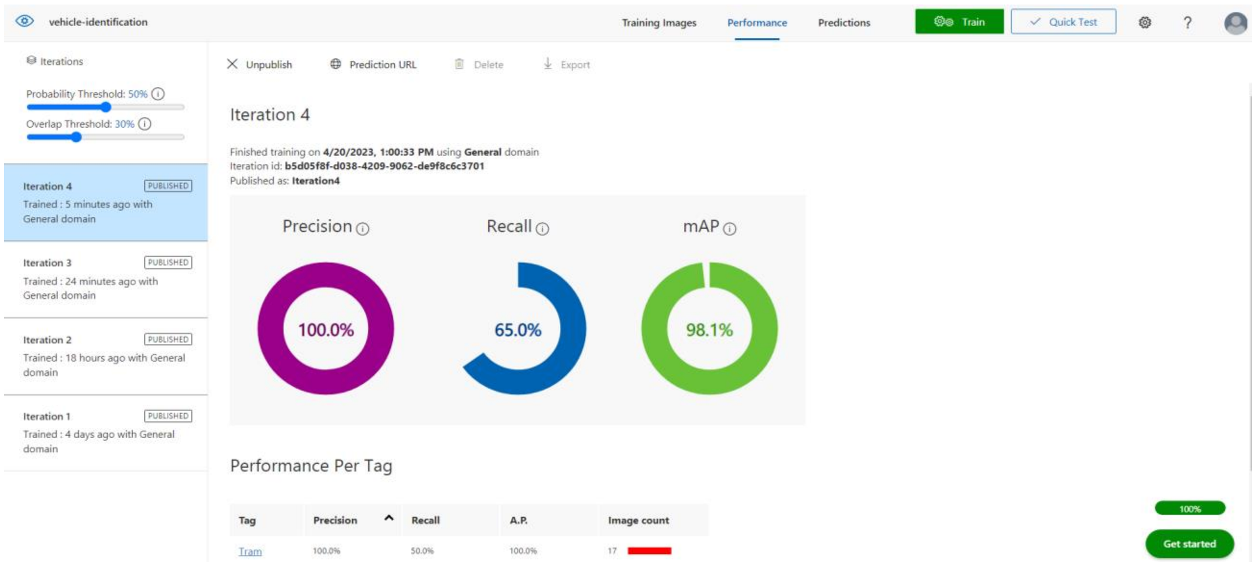




We've established a project called "**vehicle-identification**" in the custom vision portal, within the specified resource group "**cvshetty001**", for this particular assignment. The initial step after project creation is to train the model using images which are uploaded from a local directory.

We've uploaded a total of **78** images of **Bicycles**, **Cars**, and **Trams** for training purposes, along with corresponding labels. The more we train the custom vision model, the more accurate its predictions will become. Once we've uploaded all the images and applied tags, we can click on the "**Train**" tab to begin the training process.

3. Publishing the Iteration



Iteration 4 PUBLISHED
Trained : 5 minutes ago with General domain

Iteration 3 PUBLISHED
Trained : 24 minutes ago with General domain

Iteration 2 PUBLISHED
Trained : 18 hours ago with General domain

Iteration 1 PUBLISHED
Trained : 4 days ago with General domain

Iteration 4

Finished training on **4/20/2023, 1:00:33 PM** using **General** domain
Iteration id: **b5d05f8f-d038-4209-9062-de9f8c6c3701**
Published as: **Iteration4**

Performance Per Tag

Tag	Precision	Recall	A.P.	Image count
Tram	100.0%	50.0%	100.0%	17

After completing the training process, it's crucial to publish the iteration on the Custom Vision portal. This is essential because it makes the trained model available for use in various applications and services that require computer vision capabilities, like object detection or image classification(Henk 2020).

Publishing an iteration also enables you to monitor the model's performance over time, track its accuracy, and make necessary updates or improvements to enhance its effectiveness. This iterative process of training, testing, and publishing a model allows you to continually refine its performance and adapt to changing conditions or requirements.

We performed four iterations with varying training images, and the **fourth** and final iteration achieved a **precision** of **100%**, recall of **65%**, and an **mAP (Mean Average Precision)** of **98.1%.** As a result, we have published this iteration to proceed with the next step of acquiring the prediction URL.

4. Get the Prediction URL

The prediction URL API allows us to make predictions using a trained Custom Vision model through an API endpoint. To use the prediction URL API, we must first retrieve the endpoint URL and prediction key from the Custom Vision portal.

Once we have the **endpoint URL** and **prediction key**, we can construct a **POST** request to send image data to the endpoint for prediction. The request should include the image data in the request body, along with the prediction key in the headers. After the model processes the request, it will return a **JSON** response containing the predicted classes and their associated confidence scores

5. Importing the libraries

```
In [1]: import requests
        import numpy as np
        import cv2
```

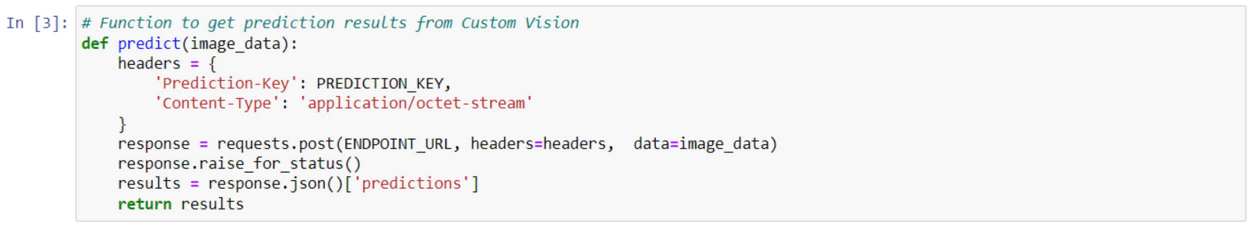
The imported code consists of three essential Python libraries. The requests library is utilized to make HTTP **requests**, **NumPy** provides powerful support for multi-dimensional arrays and matrices, and **cv2** is a feature-rich library for image and video processing that includes functions such as image and video I/O, filtering, and object detection.

6. **Set the Custom Vision Endpoint URL and Prediction key**

The code initializes two variables in Python – “**ENDPOINT_URL**” which contains a URL pointing to a Custom Vision endpoint, which is a service offered by Microsoft Azure for building and deploying custom image recognition models. Specifically, the URL refers to the detection endpoint of a particular custom vision project, iteration, and model, responsible for detecting objects within an image when submitted.

Meanwhile, “**PREDICTION_KEY**” is an API key that serves as an authentication mechanism for making requests to the Custom Vision endpoint, ensuring that only authorized entities can make predictions using the custom vision model.

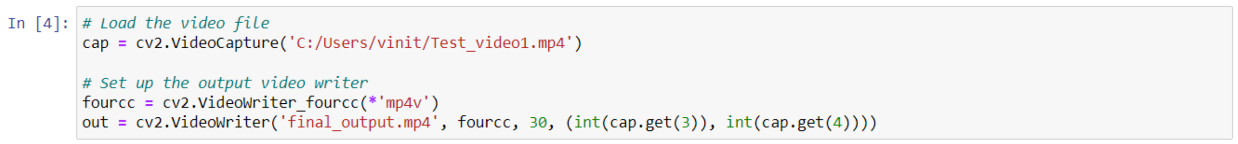
7. **Define a function to get prediction results from Custom Vision**



```
In [3]: # Function to get prediction results from Custom Vision
        def predict(image_data):
            headers = {
                'Prediction-Key': PREDICTION_KEY,
                'Content-Type': 'application/octet-stream'
            }
            response = requests.post(ENDPOINT_URL, headers=headers,  data=image_data)
            response.raise_for_status()
            results = response.json()['predictions']
            return results
```

The provided code defines a Python function named "**predict**" that uses the requests library to send an **HTTP POST request** to a Custom Vision endpoint, which is specified by the **ENDPOINT_URL** variable. The function takes an **image_data** argument and returns the prediction result in the form of a list of dictionaries. To authenticate the request, the API key and binary image data are included in the request headers. The JSON response is then parsed, and the prediction result is extracted by accessing the **'predictions'** key (Alessandro 2018).

8. **Load the video and define the output path**



```
In [4]: # Load the video file
        cap = cv2.VideoCapture('C:/Users/vinit/Test_video1.mp4')

        # Set up the output video writer
        fourcc = cv2.VideoWriter_fourcc(*'mp4v')
        out = cv2.VideoWriter('final_output.mp4', fourcc, 30, (int(cap.get(3)), int(cap.get(4))))
```

The code loads a video file located at “**C:/Users/vinit/Test_video1.mp4**” and stores it in the “**cap**” variable and sets up an output video writer. The output video writer uses the mp4v codec and writes to a file called “**final_output.mp4**”. The video is written with a frame rate of 30 fps and the dimensions of the output video are set to be the same as the dimensions of the input video which are set by the functions **cap.get(3) and cap.get(4).**

9. Vehicle detection for the trained model.

```
In [5]: # Loop through each frame of the video
        while True:
            ret, frame = cap.read()
            if not ret:
                break

            # Convert the frame to a byte array
            _, img_encoded = cv2.imencode('.jpg', frame)
            image_data = img_encoded.tobytes()

            # Get prediction results for the current frame
            results = predict(image_data)
            # Loop through the results and draw bounding boxes around detected vehicles
            for result in results:
                if result['probability'] > 0.2:
                    tag_name = result['tagName']
                    if tag_name == 'Bicycle' or tag_name == 'Car' or tag_name == 'Tram':
                        x = int(result['boundingBox']['left'] * cap.get(3))
                        y = int(result['boundingBox']['top'] * cap.get(4))
                        w = int(result['boundingBox']['width'] * cap.get(3))
                        h = int(result['boundingBox']['height'] * cap.get(4))
                        cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 0, 255), 2)
                        label = result['tagName']
                        confidence = result['probability']
                        text = f"{label} ({confidence:.2f})"
                        cv2.putText(frame, text, (x, y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

            # Write the annotated frame to the output video
            out.write(frame)
            # Show the annotated frame in a window
            cv2.imshow('frame', frame)
            if cv2.waitKey(1) == ord('q'):
                break
```

This code is designed to process a video file by looping through each frame and performing object detection on each frame using a custom vision model. It starts by reading the video file using OpenCV's VideoCapture object and setting up an output video writer to save the annotated frames to a new video file.

For each frame in the video, it converts the frame to a byte array and sends it to the predict function to obtain object detection results. It then loops through the results and draws bounding boxes around the detected vehicles whose **probability** of detection is greater than **0.2**. The label and confidence score for each detection is displayed using OpenCV's putText function.

The coordinates for the bounding boxes are calculated using the dimensions of the detected object's bounding box, which are provided in the Custom Vision API's response. Specifically, the values for the bounding box's left, top, width, and height are used to calculate the (x,y) coordinates of the bounding box in the video frame.

Furthermore, the tags used in this code to detect vehicles are “**Bicycles**”, “**Car**”, and “**Tram**”. These tags are part of a custom vision model that has been trained to recognize these specific objects in images. When the predict function is called with an input frame, it returns a list of predictions that contains information about the objects detected in the frame, including their bounding box dimensions, class label, and probability score.

The annotated frame is written to the output video file and displayed in a window using OpenCV's imshow function. The program will continue to loop through the video frames until the end of the video or until the user presses the “**q**” key to exit the program.

10. Releasing resources after object detection

```
In [6]: # Release the resources
        cap.release()
        out.release()
        cv2.destroyAllWindows()
```

The code above uses the **cv2.release()** function to release the resources utilised by the video stream and output video file after looping through each frame of the input video and finding objects inside the frames.

Finally, it uses **the cv2.destroyAllWindows()** function to close every window that the programme has opened. Utilising these features is crucial to preventing memory leaks and ensuring the effective utilisation of system resources.

11. **Video Output**

    https://drive.google.com/file/d/1IocJfV5mJ5IROOj3ZVclb7cBAw_wPzhR/view?usp=sharing

    For this project, a one-minute video was utilized as input to perform vehicle detection and tracking. The video contained a variety of vehicles such as bicycles, trams, and cars.

    To annotate the detected vehicles, bounding boxes were drawn around them. The colour used for the **bounding boxes** was **red**, which made it easier to identify the detected vehicles. Additionally, the **labels** indicating the type of vehicle detected and their **confidence score** were displayed in **green colour**. This made the labels more distinct from other visual elements in the video and easier to read. The confidence score indicated the level of certainty of the model about the presence of the detected vehicle in the frame.

    The vehicle detection and tracking performed on the video provided detailed information about the types of vehicles present in the video and their movements. This analysis could be beneficial for traffic analysis, monitoring traffic patterns, and improving traffic flow management in urban areas.