**Introduction**

**Imaging**

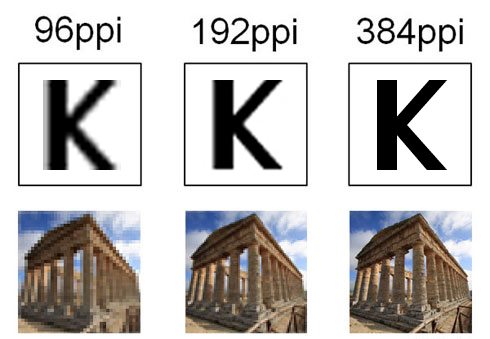
An image is a visual representation that can be either two-dimensional, such as a drawing, painting, or photograph, or three-dimensional, like a carving or sculpture.

* Image**:** A visual representation in the form of a function f(x,y) where f is related to the brightness (or color) at point (x,y).
* Most images are defined over a rectangular region and are continuous in both amplitude and space.

**Digital Images and Pixels**

Digital Image is called a set of discrete samples f[x,y] representing a continuous image f(x,y).

Each element of the 2D array f[x,y] is called a pixel or pel (short for “picture element”).



**Distinctions Between Computer Vision and Image Processing**

In image processing, the input is an image, and the output is an image as well. Conversely, in computer vision, an image or video is taken as input, and the output could be an enhanced image, an understanding of the content, or even the behavior of a computer system based on this understanding.

Image Processing: Visual Perfection

The primary aim of image processing is to improve image quality. Whether it’s enhancing contrast, adjusting colors, or smoothing edges, the focus is on making the image more visually appealing or suitable for further use. It involves transforming the raw image into a refined version of itself.

Image processing is essential in fields like digital photography for color correction, medical imaging for clearer scans, and graphic design for creating stunning visuals. These transformations not only improve aesthetics but also make images more suitable for analysis, laying the groundwork for deeper interpretation, including by computer vision systems.

Computer Vision: Decoding the Visual World

Computer vision seeks to extract meaning from images. The goal isn’t to change how the image looks but to understand what the image represents. This involves identifying objects, interpreting scenes, and recognizing patterns and behaviors within the image. It focuses on comprehension rather than alteration.

**Color Representation**

In digital image processing, colors are typically represented using various color models. Color models define how colors can be represented using different components. Common color models include RGB (Red, Green, Blue), CMYK (Cyan, Magenta, Yellow, Black), and HSV (Hue, Saturation, Value).

8-Bit Grayscale Image

An 8-bit grayscale image is a digital picture composed of tiny dots called pixels. Each pixel can display one of 256 different shades of gray. The "8-bit" term refers to the amount of information each pixel can hold, which in this case is 8 bits. This allows each pixel to store a number from 0 to 255, representing the shade of gray for that particular pixel.

8-Bit Color Image

An 8-bit color image has three color channels (red, green, and blue), each with 8 bits. The total number of bits per pixel is 24 (8 bits/channel × 3 channels). Each color channel can have 256 intensity levels, ranging from 0 (minimum intensity) to 255 (maximum intensity). This allows for 256×256×256=16,777,216 possible color combinations, defining the RGB channels. This results in a lower color depth compared to images with higher bit depths, such as 24-bit (true color) or 32-bit images.

Before going to the practicle have to create a virtual environment.

**Creating Virtual Environment**

A virtual environment in computing, particularly in the context of software development, is an isolated environment that allows developers to run and manage project-specific dependencies independently from the system-wide packages. This isolation helps in avoiding conflicts between different projects' dependencies and ensures that each project has access to the specific versions of libraries and tools it requires.

In Python, virtual environments are commonly used, and there are several tools available to create and manage them. The most widely used tool is venv, which comes built-in with Python 3.3 and later versions. Another popular tool is virtualenv, which provides more features and is compatible with older versions of Python.

**Creating a Virtual Environment using venv**

1. Go to the respective folder structure where you want to create the environment
2. Executing the bellow command

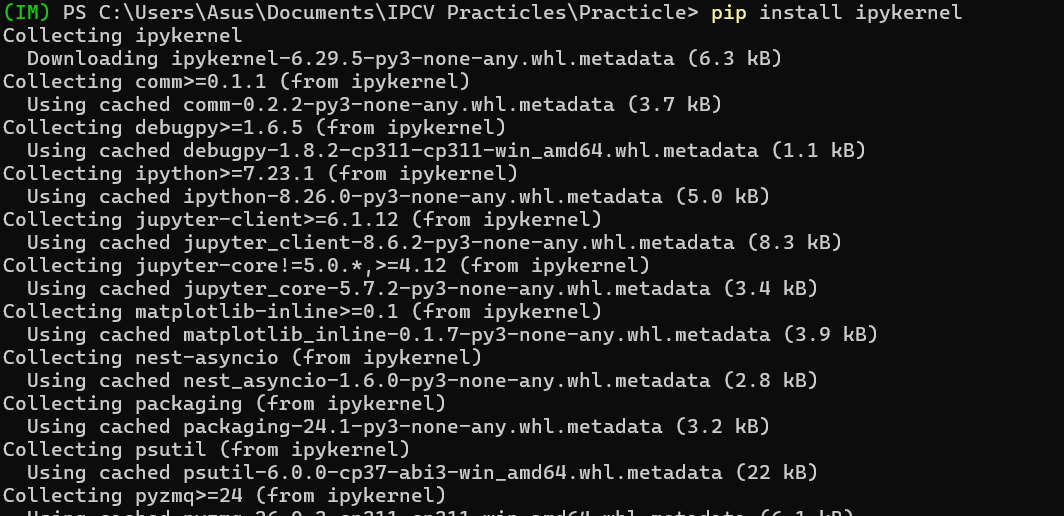


This command creates a virtual environment named myenv in the current directory

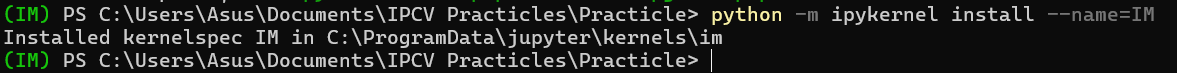
1. Activate the respective virtual environment that has been created following the steps below.



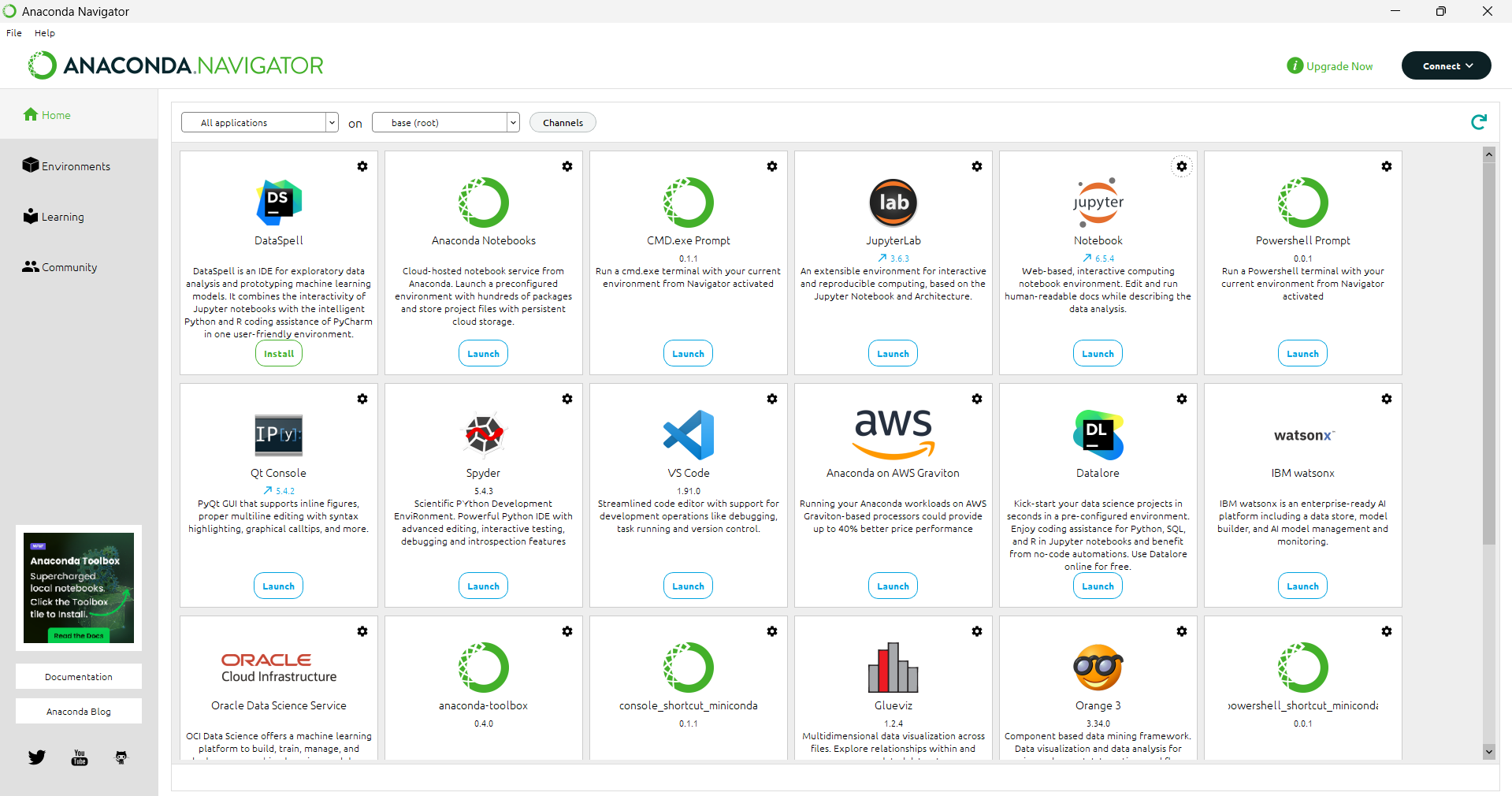
1. After activating the relevant environment next required Kernal has to be created. Kernel will link the customized package assembly Jupiter as relevant environment.
2. Execute the below command to activate the Kernal installation



1. Once the Kernal is installed execute the command for Kernal linkin development environment.



1. Then go to the Jupiter environment and select the own path and then Kernal



A screenshot of a computer

Description automatically generated

**Practicle 01: Shape Detection**

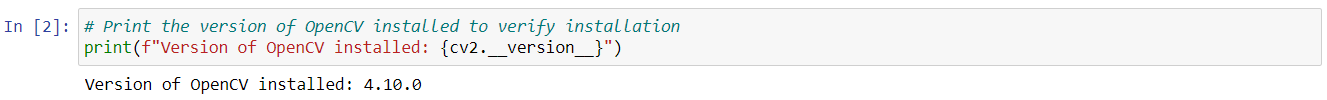
Firstly, we have to import the necessary libraries and check if OpenCV is installed correctly.

Therefor we use below libraries.

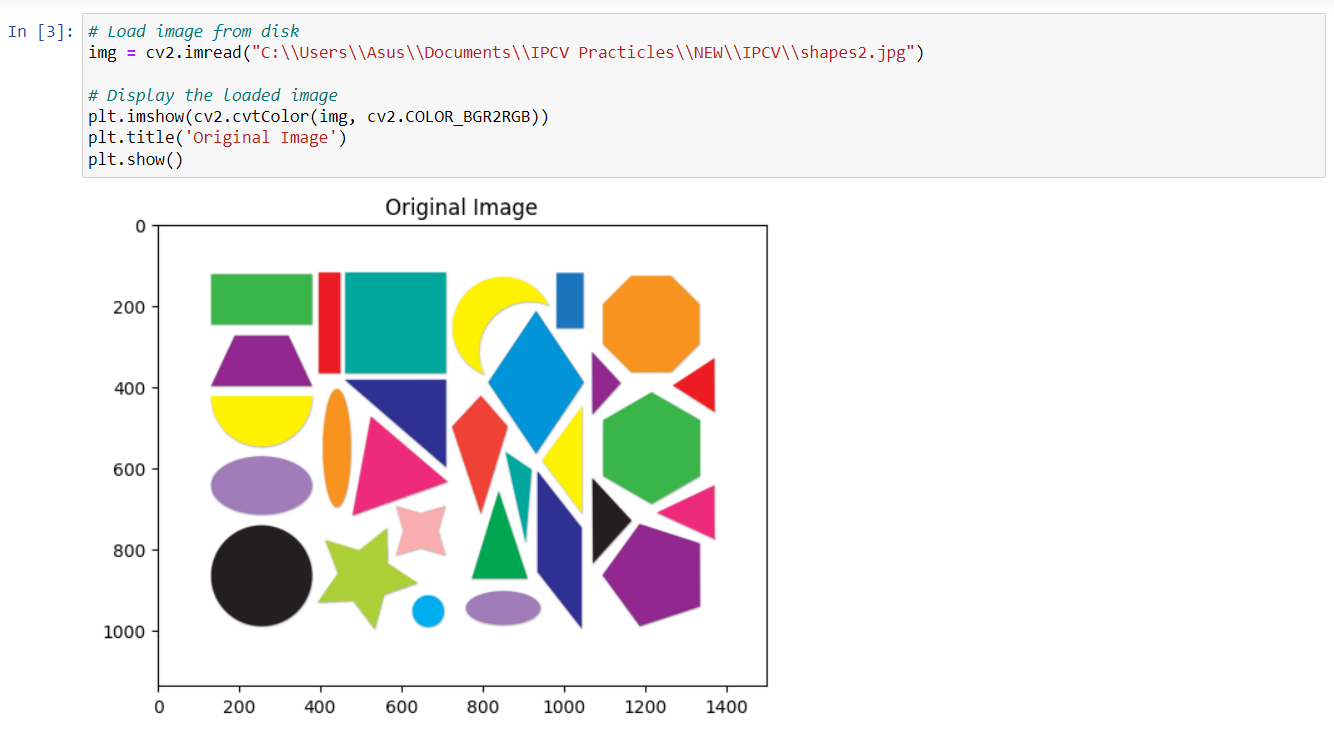
* OpenCV – use for computer vision tasks
* Numpy – use for array operations
* Matplotlib – use for plotting



Then I can check whether OpenCV-python has been installed properly in my virtual environment.



Now I can load the image to be processed from my disk.

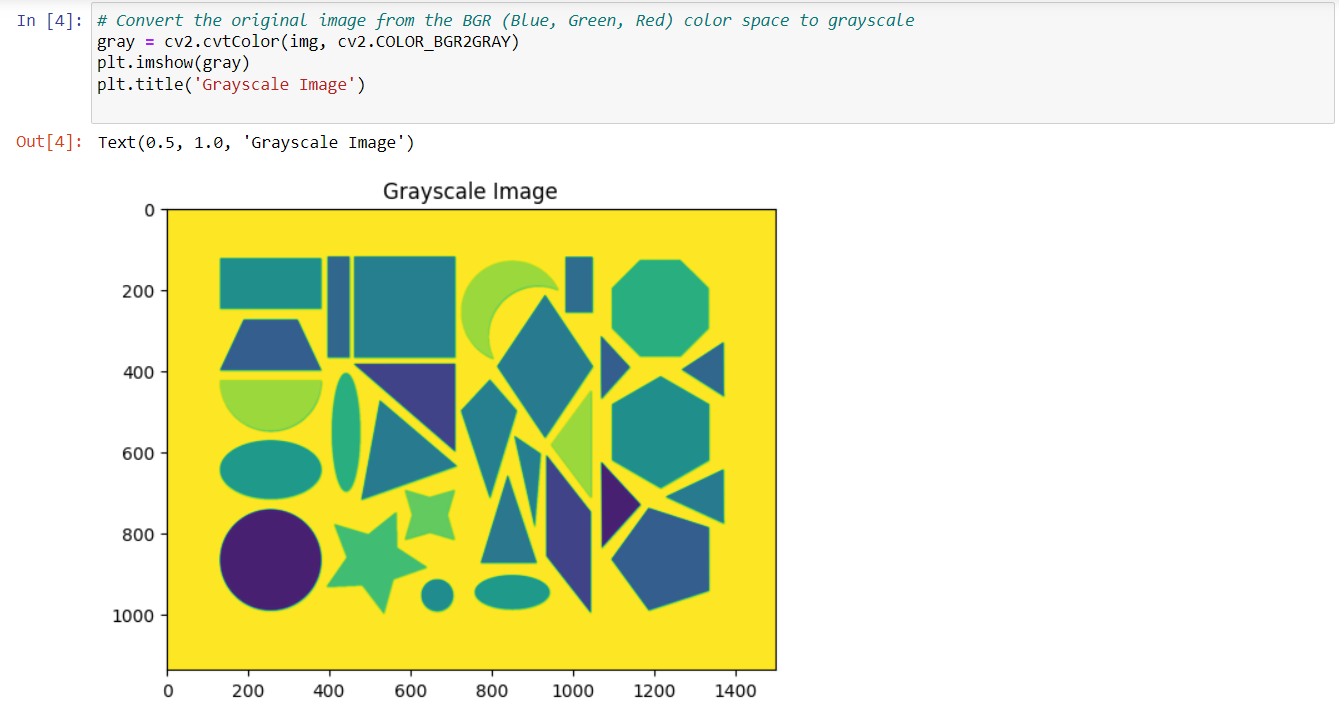


Here, I use the cv2.imread() function to load the image. This function returns a Numpy array, which is now stored in the img variable. Since OpenCV uses the BGR (Blue, Green, Red) color space by default and Matplotlib uses the RGB color space, I convert the image to RGB for correct color display using cv2.cvtColor(img,cv2.COLOR\_BGR2RGB).

Then I can do the image preprocessing part.

1. **Grayscale Conversion**

A common step in preprocessing during image processing is converting images to grayscale. Grayscale images only have one channel, which stores the intensity (brightness) of each pixel.



I convert the image to grayscale because:

* **Simplicity**: Grayscale images are simpler to process than color images, since they only have one value (intensity) per pixel, instead of three values (R, G, B) as in color images.
* **Reduced Complexity**: The simplicity of grayscale images results in requiring less computational power (less memory and time consumption) to process them.
* **Feature Extraction**: Grayscale images are more effective for algorithms that require feature extraction and analysis, such as edge detection, object recognition, etc.

1. **Gaussian blur**

Gaussian blur is a type of image-blurring filter that uses a Gaussian distribution to calculate the weighted average of neighboring pixels

A screenshot of a computer screen

Description automatically generated

The (5, 5) argument passed into the GaussianBlur() function denotes the kernel size, which is the size of the "paintbrush" used to blend the pixels together. A larger "paintbrush" (larger kernel) means more blending. The 0 argument denotes the standard deviation (σ) in the Gaussian distribution.

1. **Edge Detection**

In here I use Canny edge detection technology to detects edges in images where rapid intensity changes can be noticed. I use the cv2.Canny() function to implement Canny edge detection.

A screenshot of a computer

Description automatically generated

The cv2.Canny() function takes in three parameters:

1. The input image
2. The lower threshold - identifies faint changes in intensity
3. The upper threshold - identifies strong changes in intensity
4. **Thresholding**

Thresholding is an image processing technique where pixel values are updated to specific values based on whether their intensity is below or above a certain threshold.

A screenshot of a computer

Description automatically generated

The cv2.threshold() function takes in:

1. The input image - must be a grayscale image
2. The threshold - pixels with intensities lower than this value will be set to 0 (black)
3. The maximum value - pixels with intensities above the threshold will be set to this value
4. The type of thresholding - cv2.THRESH\_BINARY for binary thresholding

**Contour Identification**

Contours are the continuous curves or boundaries that form the outlines of objects in an image.

A screen shot of a computer

Description automatically generated

The cv2.findContours() function identifies and returns contours in a binary image. It takes in:

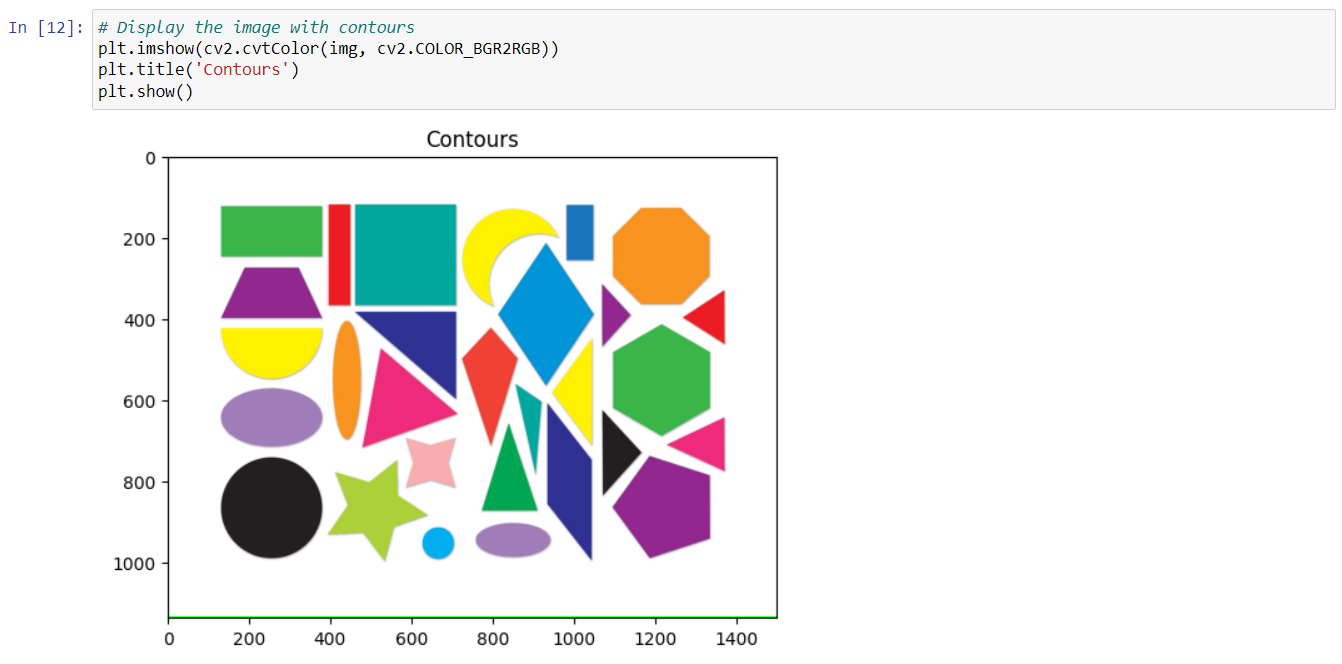
1. The input image - must be a binary image
2. The retrieval mode - e.g., cv2.RETR\_EXTERNAL retrieves only the outermost contours
3. The approximation method - e.g., cv2.CHAIN\_APPROX\_SIMPLE compresses horizontal, vertical, and diagonal segments and leaves only their end points

**Drawing Contours**

A screenshot of a computer

Description automatically generated

Then display those contours in original image.

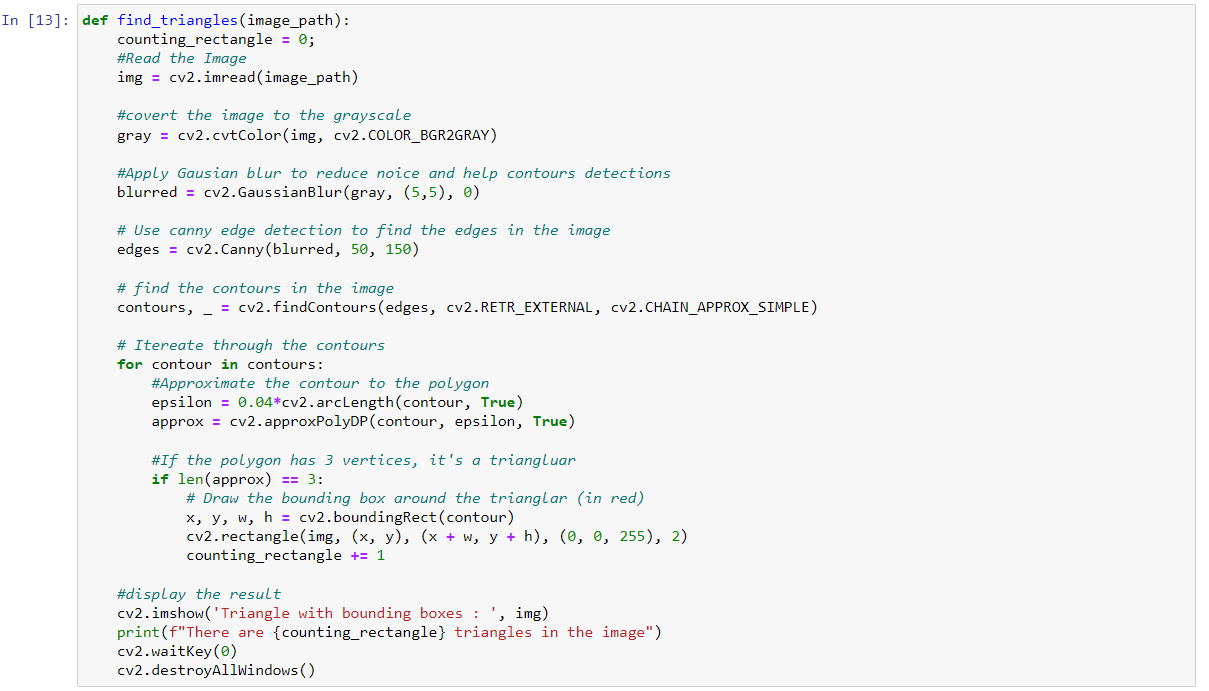


The cv2.drawContours() function takes in:

1. The input image
2. The list of contours obtained
3. The index of the contour(s) to be drawn - -1 to draw all contours
4. The color of the contours - (0, 255, 0) for green
5. The thickness of the contour lines

**Shape Detection: Finding Triangles**

Using the above-mentioned preprocessing steps, I can now identify shapes in an image. Here's a function to detect triangles in an input image.



In here, epsilon as deciding how much you care about the tiny details of the shape. Smaller epsilon means you really care about the details, while larger epsilon means you're okay with a simpler, more generalized shape.

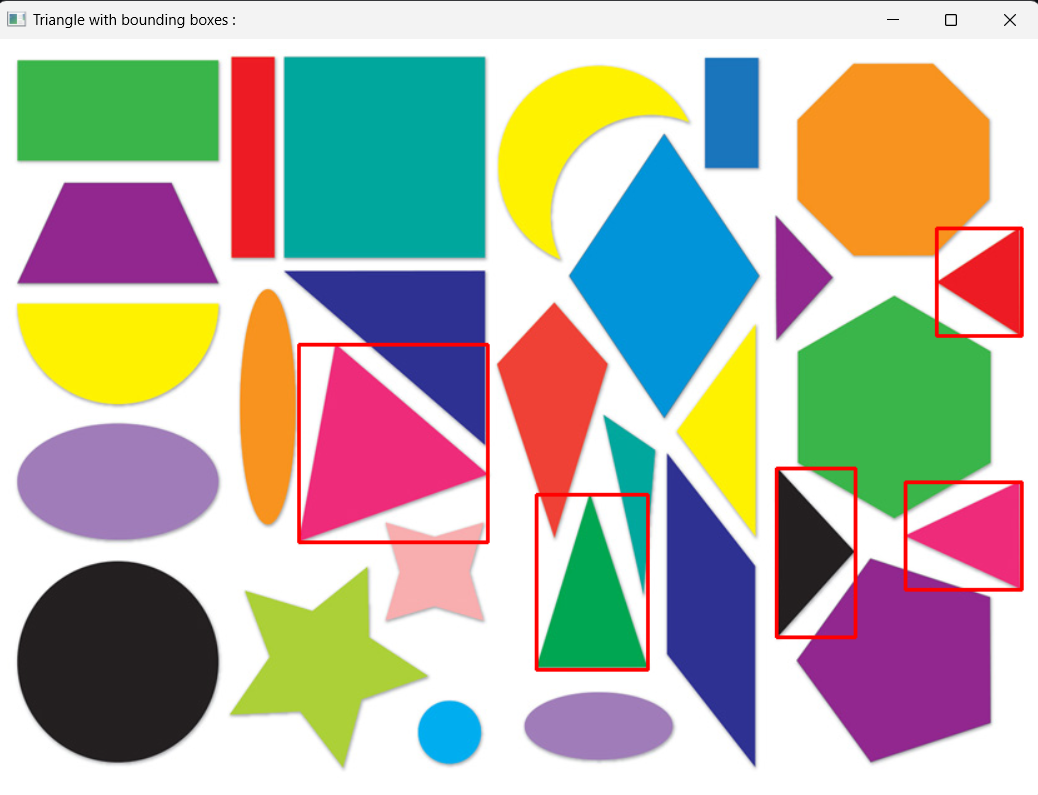
archLength() method calculate the total lenth of the surface contour and 4% of is applied for the Epsilon calculation.

True denotes, we are looking for closed shapes, not open. appoxPolyDP method derives information of all poligon shapes, which satisfies the specified epsilon criteria.

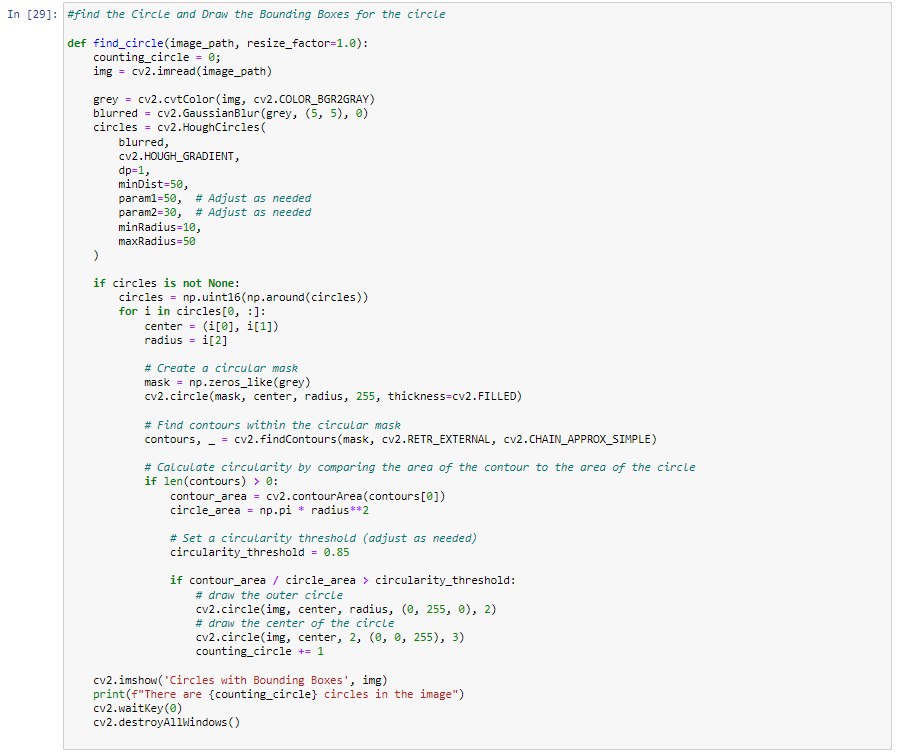
Henceforth, via a condition we can check poligons having 3 contour lines connected as traingles. If you make it 4, it will look for squares.

Then I call the function to detect triangles in the image.

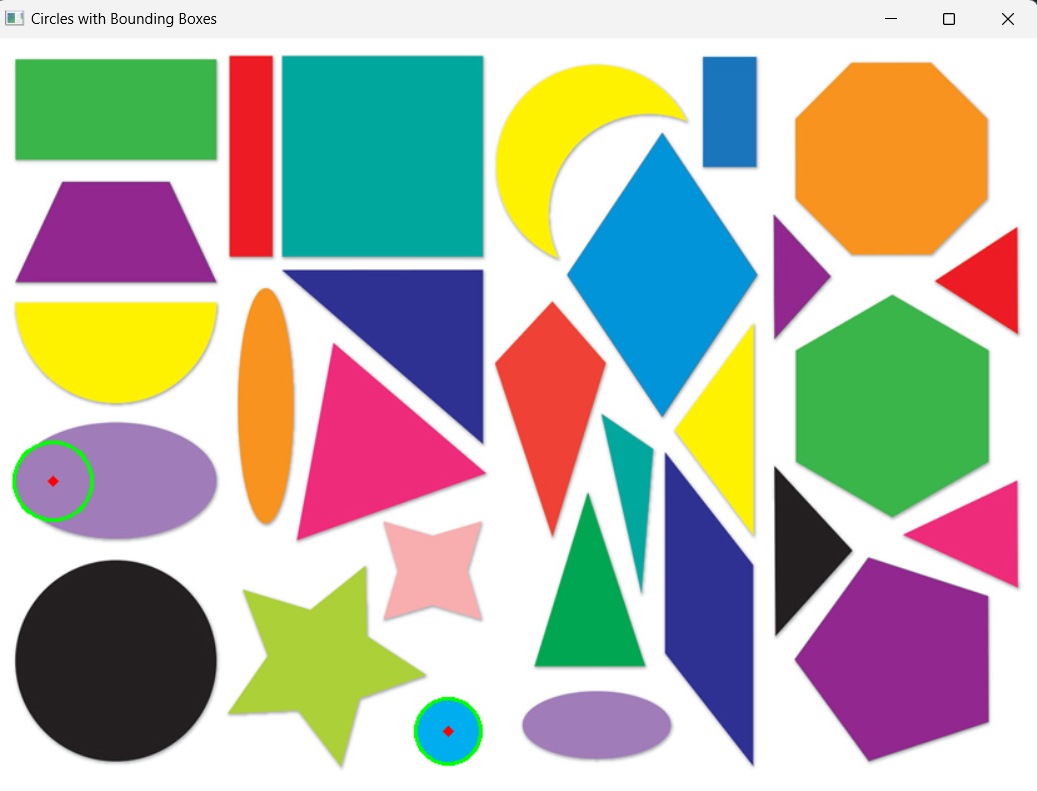
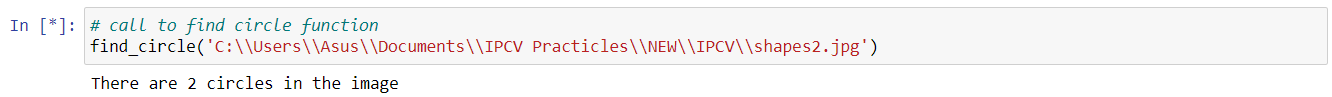




**Tried to develop a function to find circles**



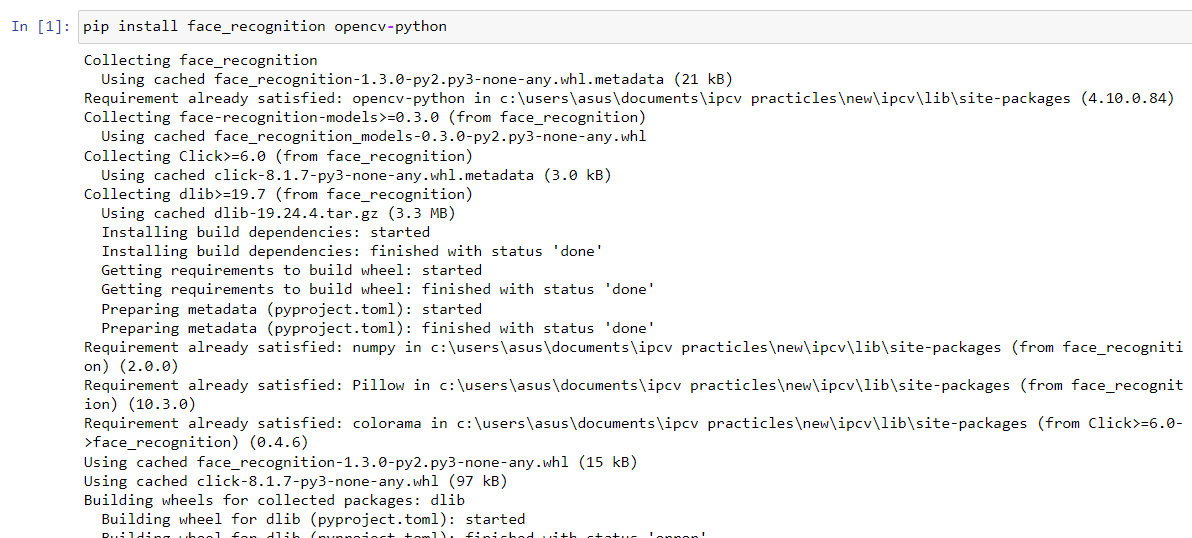
Then I call the function to detect circles in the image.



**Practical 02: Intelligent Template Matching**

Firstly, we have to import the necessary libraries.

Therefor we use below libraries.



This command installs the OpenCV library for image processing, the face\_recognition library for face detection and recognition.

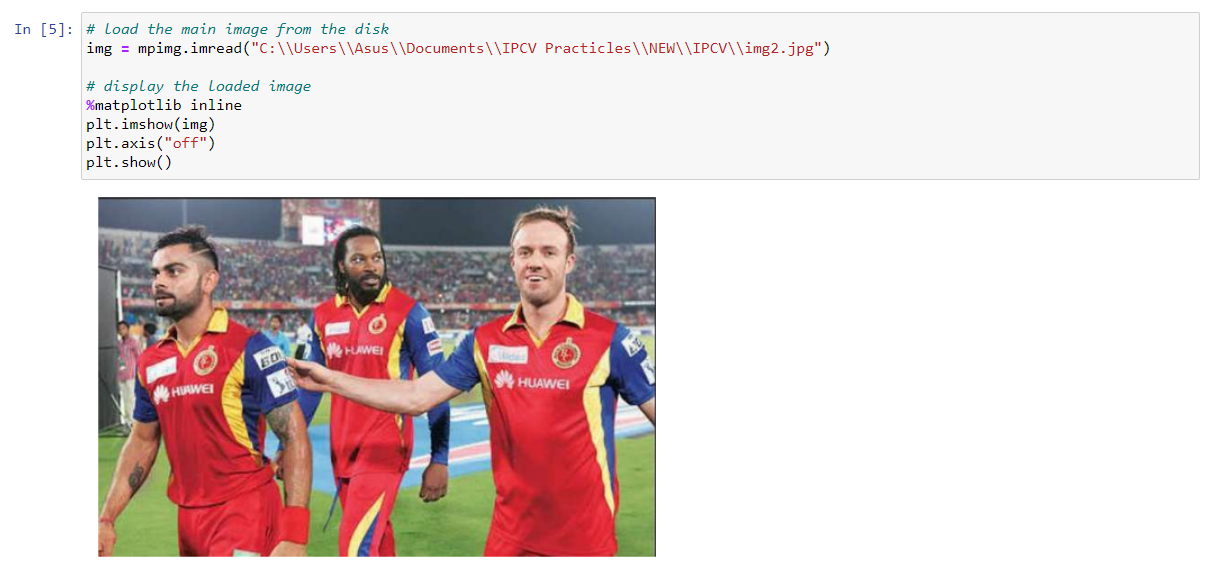


* face\_recognition is used for detecting and encoding faces.
* cv2 from OpenCV is used for image processing.
* numpy is used for handling arrays.
* matplotlib.pyplot and matplotlib.image are used for displaying images.

Here we have to use two images called target image and main image. The target image is the image we recognize in the main image.

**Load and Display Images**

Load the main image containing multiple faces.



%matplotlib inline ensures that plots are displayed within the notebook. plt.imshow(img) displays the image stored in img and plt.axis("off") removes the axes for a cleaner display.

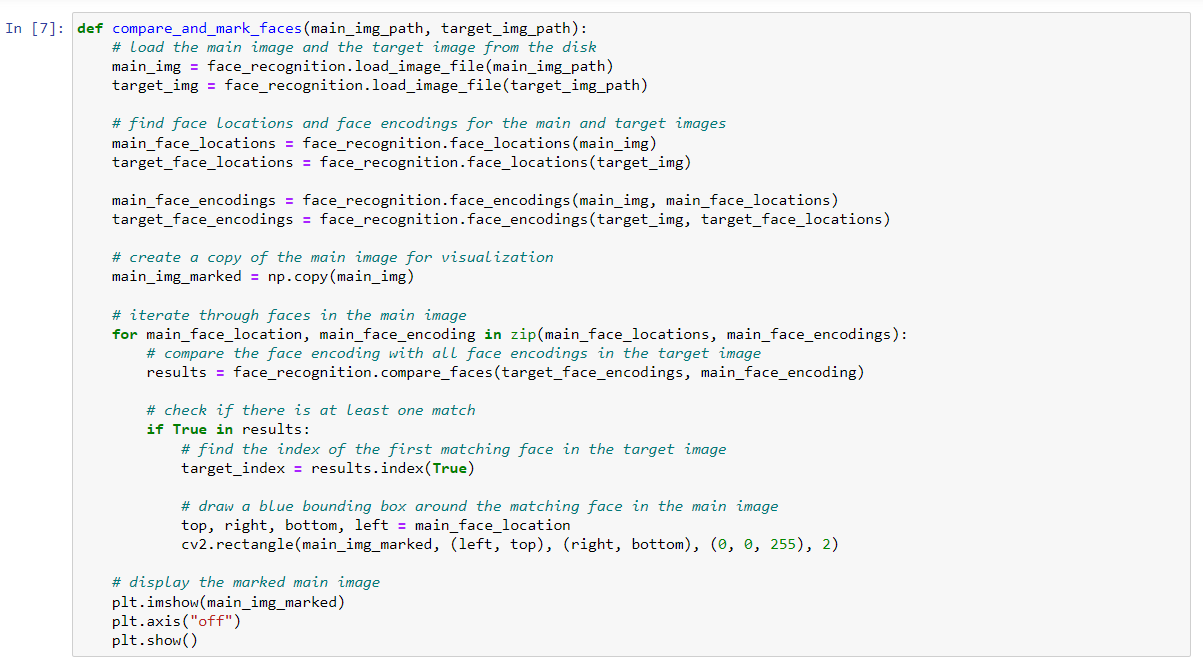
plt.show() renders the image.

Then I load the target image containing the face to be identified.



**Face Recognition**

Define a function to compare and mark faces in the main image



First line defines a new function named compare\_and\_mark\_faces that takes two arguments: main\_img\_path and target\_img\_path, which are the file paths of the main and target images, respectively. use face\_recognition.load\_image\_file to load the images from the given file paths.

face\_recognition.face\_locations returns the coordinates of all faces found in the image. face\_recognition.face\_encodings generates unique encodings for each face based on their features. The face locations and encodings for both the main and target images are stored in their respective variables.

This main\_img\_marked = np.copy(main\_img) line creates a copy of the main image array named main\_img\_marked to avoid altering the original image.

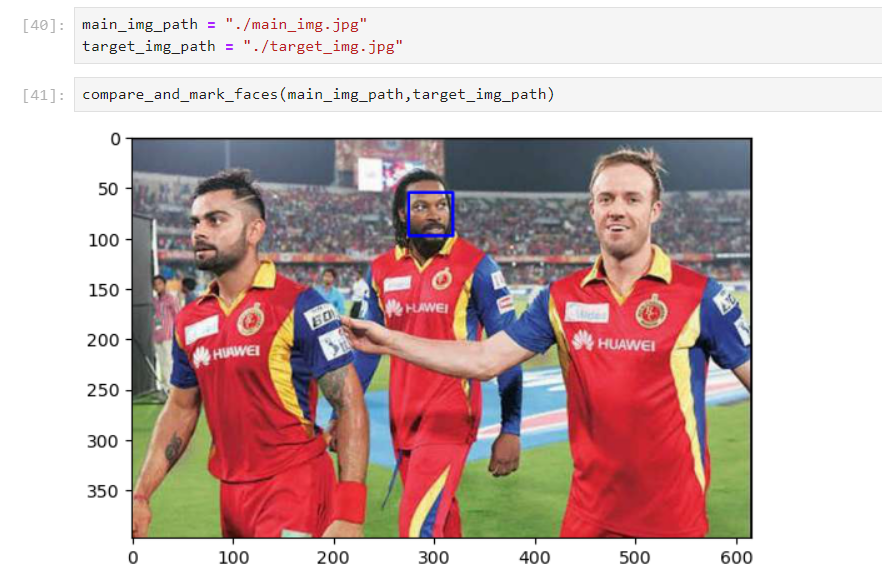
Then starts a loop that iterates over each face location and encoding in the main image using the zip function to pair them together. results = face\_recognition.compare\_faces(target\_face\_encodings, main\_face\_encoding) line compares the encoding of the current face in the main image with the encodings of all faces in the target image. It returns a list of Boolean values indicating whether a match was found.

The line if True in results: checks if any match was found in the comparison results. target\_index = results.index(True) finds the index of the first matching face in the target image.

Then I need to draw a rectangle around the matching face in the main image. Unpack the coordinates of the matching face location into top, right, bottom, and left. Then cv2.rectangle draws a rectangle around the face using these coordinates. The color of the rectangle is blue (0, 0, 255), and the thickness is 2 pixels.

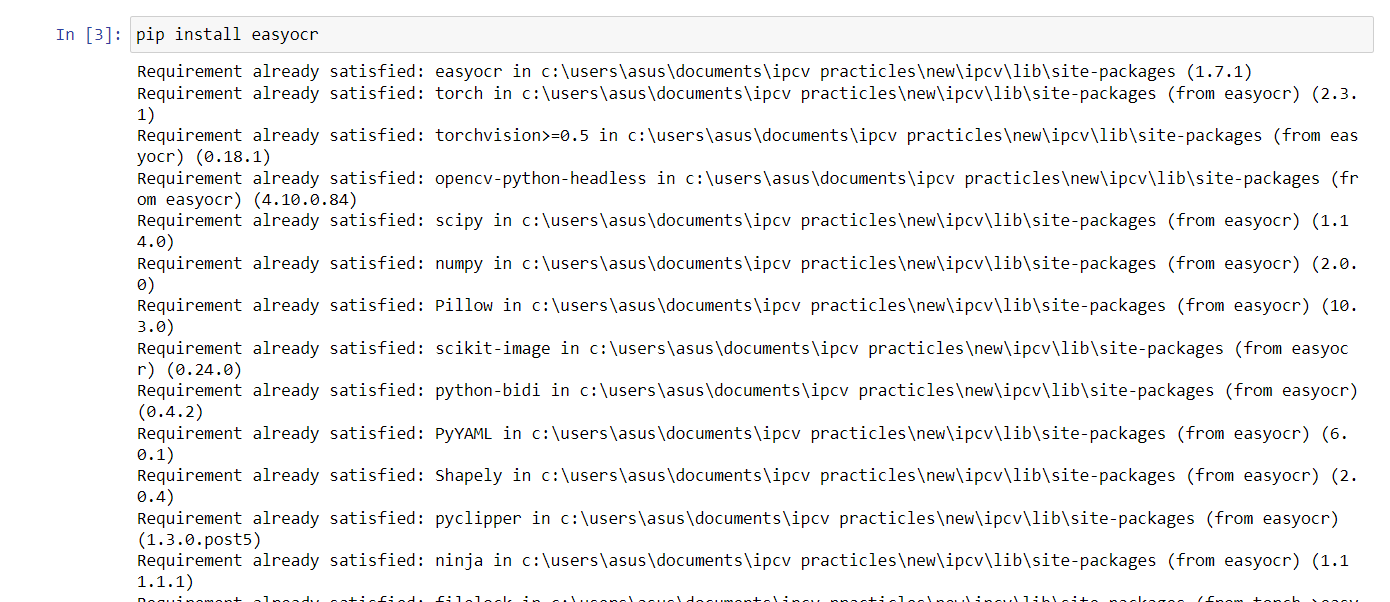
These lines display the marked main image similarly to how the original images were displayed.

Finally, I can call this function with the two images of Timothée Chalamet to see how well it works.



**Practical 03: Vehicle Number Plate Detection**

In this practicle we can use 2 libraries called Tesseract OCR and EasyOCR. Tesseract can develop customized setup. EasyOCR is good for basic detections. Therefore I used EasyOCR for this practicle.



First, I import the necessary libraries for this practicle.



* cv2: This is the OpenCV library, used for image processing tasks.
* easyocr: This library is used to perform Optical Character Recognition (OCR) to extract text from images.
* matplotlib.pyplot: This is used to display images and results graphically.
* matplotlib.image: This module within matplotlib is used for reading images.

Next, I load the image of the number plate from disk and display it.



mpimg.imread("./img/number-plate-only.jpg") function loads the image located at the specified path and %matplotlib inline ensures that the images are displayed within the Jupyter notebook. plt.imshow(img)displays the loaded image and plt.axis("off")removes the axes from the displayed image. Then plt.show()renders the image on the notebook.

**Image preprocessing part**

I define a function to preprocess the image. This involves converting it to grayscale and applying a Gaussian blur.

A white background with black text

Description automatically generated

img = cv2.imread(img\_path)reads the image from the specified file path and gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)converts the image from BGR (Blue, Green, Red) color space to grayscale. blurred = cv2.GaussianBlur(gray, (5, 5), 0)applies a Gaussian blur to the grayscale image to reduce noise. The kernel size (5, 5) determines the intensity of the blur. return blurred returns the blurred grayscale image.

**Perform OCR**

Next, I define a function to perform OCR on the preprocessed image.

A close-up of a computer screen

Description automatically generated

result = reader.readtext(img)uses the EasyOCR reader to read text from the image. recognized\_text = ' '.join([entry[1] for entry in result])extracts the text from the OCR result. entry[1] is the recognized text in each OCR result entry. The texts are joined with a space to form a single string. return recognized\_text returns the recognized text as a single string.

**Displaying Results**

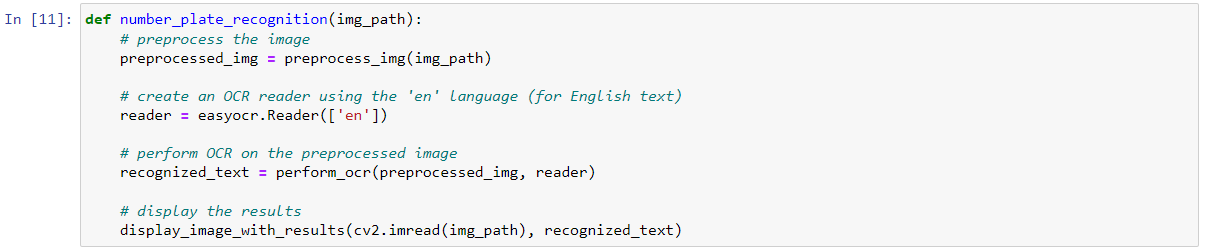
Then, I define a function to display the results along with the input image.

A white background with a black and blue text

Description automatically generated with medium confidence

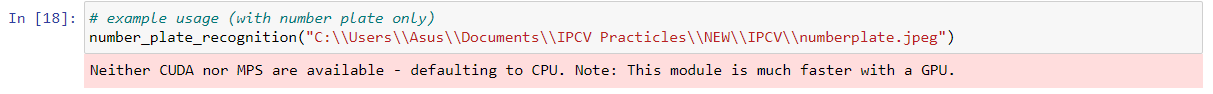
plt.imshow(cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)) converts the image from BGR to RGB color space (as OpenCV uses BGR by default) and displays it. plt.title(f"Recognized Text: {text}")sets the title of the image plot to the recognized text. plt.axis("off") removes the axes from the plot and plt.show()renders the plot with the image and title.

Finally, I define a function that integrates all the previous steps into a single operation for performing number plate recognition.

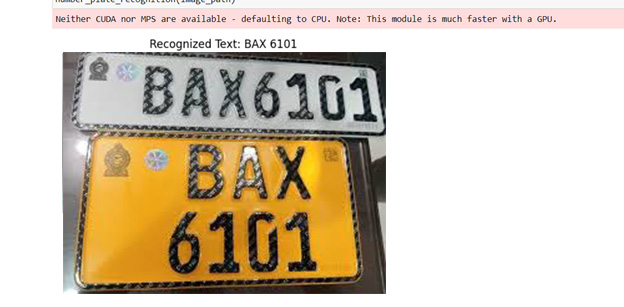


preprocessed\_img = preprocess\_img(img\_path) calls the preprocessing function to get the preprocessed image and reader = easyocr.Reader(['en']) creates an EasyOCR reader object configured for English text. recognized\_text = perform\_ocr(preprocessed\_img, reader)performs OCR on the preprocessed image to get the recognized text. display\_image\_with\_results(cv2.imread(img\_path), recognized\_text)displays the original image along with the recognized text.

Finally, I use the defined function to perform number plate recognition on an example image.

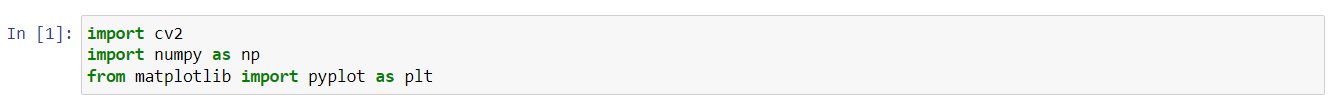


number\_plate\_recognition("./img/number-plate-only.jpg") calls the function with the path to the number plate image, performing the entire OCR process and displaying the results.



Sometimes this kind of numberplates can’t recognize in this method. In such cases region of interest may be more effective.

**Practical 04: Vehicle Number Plate Analysis with Region of Interest (Dumb Method)**



* cv2 is the OpenCV library used for computer vision tasks.
* Numpy is used for numerical operations, especially on arrays.
* pyplot from matplotlib is used for plotting images.

**Region of Interest (ROI)**

Region of Interest (ROI) is a technique used in image processing to focus on specific areas of an image for analysis. This can be particularly useful when you want to perform operations on a particular part of an image, such as detecting objects, analyzing patterns, or extracting specific features. By isolating the ROI, you can reduce computational complexity, improve processing speed, and enhance accuracy by excluding irrelevant parts of the image.

An ROI is essentially a subset of the image that I want to analyze separately from the rest of the image. This region can be defined in various shapes, such as rectangles, circles, polygons, or any arbitrary shape. The most common method is to use a rectangular region.

A computer code with many small text

Description automatically generated with medium confidence

Firstly, defining the find\_number\_plate Function and it takes a single argument, image\_path, which is the path to the image file containing the car.

After that I move to the Image loading and preprocessing part. cv2.imread(image\_path)reads the image from the specified path. cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)converts the image from BGR color space to grayscale. This is a common preprocessing step in image processing. cv2.GaussianBlur(gray, (3,3), 0)applies Gaussian blur to the grayscale image to reduce noise and detail, which helps in edge detection.

cv2.Canny(blurred, 50, 450) uses the Canny edge detection algorithm to find edges in the blurred image. The parameters 50 and 450 are the thresholds for the hysteresis procedure.

Then I find the contours. For that cv2.findContours(edges, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE) finds the contours in the edge-detected image. cv2.RETR\_EXTERNAL retrieves only the external contours, and cv2.CHAIN\_APPROX\_SIMPLE compresses horizontal, vertical, and diagonal segments and leaves only their end points.

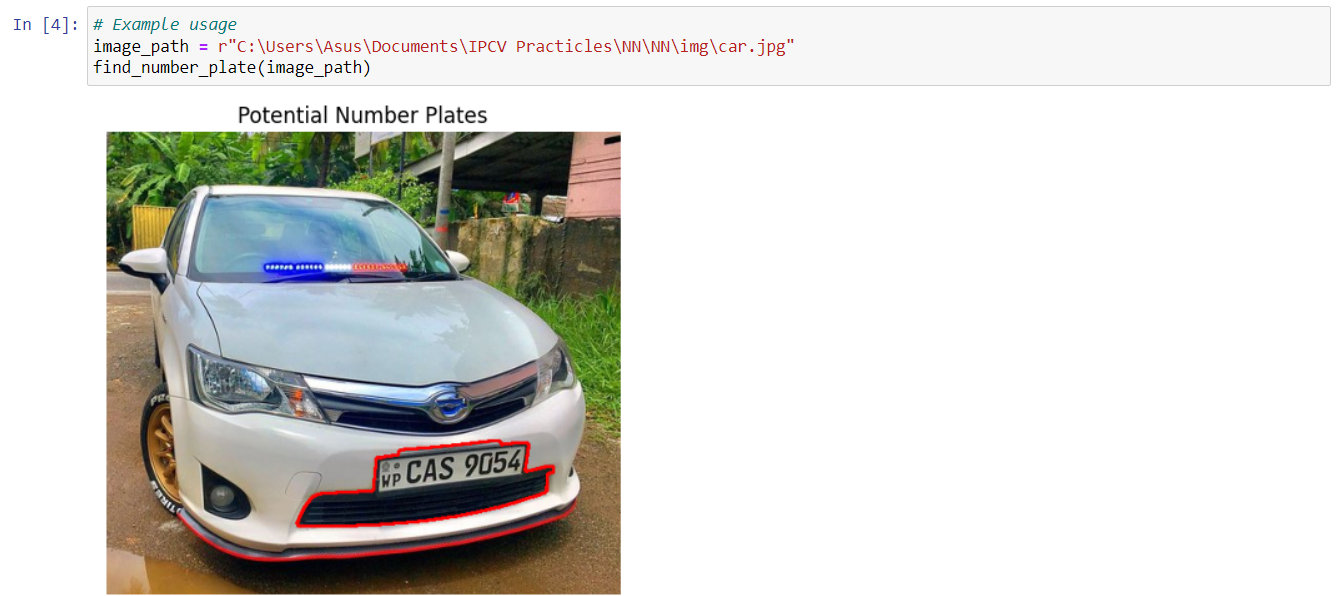
After that Initializes an empty list potential\_plates to store contours that might correspond to number plates. Iterates through each contour found cv2.contourArea(contour) calculates the area of the contour. if 1000 < area < 50000 filters the contours based on their area. The chosen range (1000 < area < 50000) is an empirical guess to filter out small or excessively large areas that are unlikely to be number plates.

plate\_img = img.copy() creates a copy of the original image to draw on.

cv2.drawContours(plate\_img, potential\_plates, -1, (0,0,255), 2) draws the filtered contours on the copied image. The color is set to red (0,0,255), and the thickness of the contour lines is 2.

For the purpose of displaying results cv2.cvtColor(plate\_img, cv2.COLOR\_BGR2RGB) use to converts the image from BGR to RGB color space, which is needed for proper display using matplotlib. plt.imshow(...)use to displays the image. plt.title("Potential Number Plates")sets the title of the plot. plt.axis('off')hides the axis. plt.show()shows the plot.

Then specifies the path to the image and calls the find\_number\_plate function to process the image.



**Practical 05: Object Detection with MobileNet**

First, I import the necessary libraries for this practicle.



* cv2 is the OpenCV library, which we use for computer vision tasks.
* matplotlib.pyplot module is part of the Matplotlib library and is used for plotting images and other graphical representations.

**MobileNet**

MobileNets are a family of deep learning models developed by Google Research. These models are designed specifically for computer vision applications on mobile and embedded devices where computational resources are limited. MobileNets are built to be lightweight and efficient while maintaining a good level of accuracy. They achieve this by using depth-wise separable convolutions, a special type of convolution operation that reduces the number of parameters and computational cost compared to traditional convolutions.

Convolutional Neural Networks (CNNs)

A type of artificial neural network designed for visual tasks like image classification and object detection. CNNs use convolutional layers to detect patterns in images.

**Depth-Wise Separable Convolutions**

Depth-wise separable convolutions are the core innovation that makes MobileNets efficient. They consist of two main operations:

1. Depth-Wise Convolution:

Instead of applying a single filter to the entire input volume, depth-wise convolution applies a separate filter to each channel of the input volume. For example, an image with three color channels (RGB) will have three separate filters applied.

This operation reduces the number of computations significantly compared to a regular convolution.

1. Point-Wise Convolution:

After depth-wise convolution, the output is combined using a 1x1 filter. This operation is known as point-wise convolution.

The point-wise convolution integrates the outputs of the depth-wise convolution by combining them into a single-channel output.

**Advantages of MobileNet**

* Reduced Parameters: Depth-wise separable convolutions significantly reduce the number of parameters in the model, which helps in reducing the model size.
* Lower Memory Requirements: With fewer parameters, MobileNets require less memory, making them suitable for mobile and embedded devices.
* Efficient Computation: The reduced number of operations in depth-wise separable convolutions leads to faster inference times.
* Reduced Overfitting: With fewer parameters, the risk of overfitting is lowered, which helps in generalizing better to new data.

**Applications of MobileNet**

* Image Classification: Identifying the class of objects in an image.
* Object Detection: Locating and identifying objects within an image.
* Semantic Segmentation: Classifying each pixel in an image into a category.
* Face Recognition: Identifying and verifying faces in images.

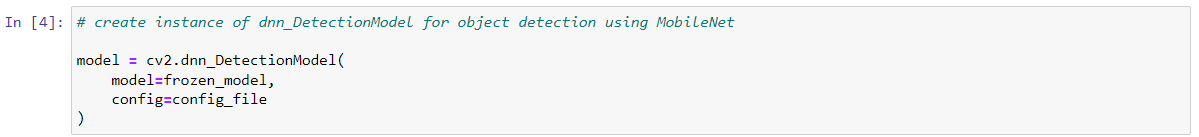
**MobileNet Variants**

There are several variants of MobileNet, each designed to balance the trade-off between accuracy and efficiency:

1. MobileNetV1: The original version, introduced depth-wise separable convolutions.
2. MobileNetV2: Introduced the concept of inverted residuals and linear bottlenecks to improve performance.
3. MobileNetV3: Further optimized using a combination of automated machine learning techniques and manual design to enhance both efficiency and accuracy.



Here I use Pre-trained models. frozen\_model is the file name of the pre-trained MobileNet model. config\_file is the configuration file containing metadata about the model.



cv2.dnn\_DetectionModel function creates an instance of the detection model using the specified model and configuration files.

A screen shot of a computer

Description automatically generated

class\_labels is An empty list to store object labels. file\_name is the file containing labels for objects. The with open block reads the file and splits its contents into a list of labels, removing any trailing newline characters.

**Preprocessing**

A white rectangular object with a black border

Description automatically generated

setInputSize(320, 320)resizes the input image to 320x320 pixels, setInputScale(1.0 / 127.5)scales the input data and setInputMean((127.5, 127.5, 127.5))centers the input data around zero by subtracting 127.5 from each pixel value. setInputSwapRB(True)swaps the red and blue color channels, as OpenCV loads images in BGR format, but MobileNet expects RGB format.

Let’s load and Display a Sample Image



cv2.imread("./img/man-and-car.jpg")loads the sample image and plt.imshow(img) displays the image using Matplotlib.

Then I need swap the color channels.



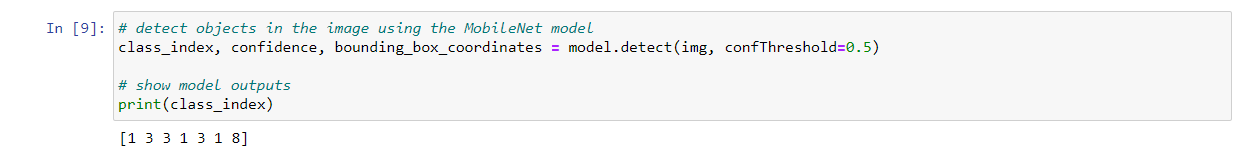
cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)converts the image from BGR to RGB format and plt.imshow(converted\_img) displays the converted image.

Finally, we can call model.detect() on the processed image to retrieve a list of objects detected in the image. This function takes in two arguments:

* The input image, converted from BGR format to RGB format.
* The confidence threshold, which is the level at which detections will be considered. For example, for a confidence threshold of 0.5, only detections having a confidence of 50% or more will be considered.

It returns three values:

* The index of the label of the object. This is an int value.
* The confidence score (i.e. probability) of the predictions made by the model.
* The coordinates of the bounding boxes to be mapped on the detected objects.



model.detect(img, confThreshold=0.5)detects objects in the image with a confidence threshold of 50%. Returns: class\_index it is the index of the detected object's label, confidence the confidence score of the detection and bounding\_box\_coordinates coordinates of the bounding boxes around the detected objects.

print(class\_index) prints the class indices of detected objects.

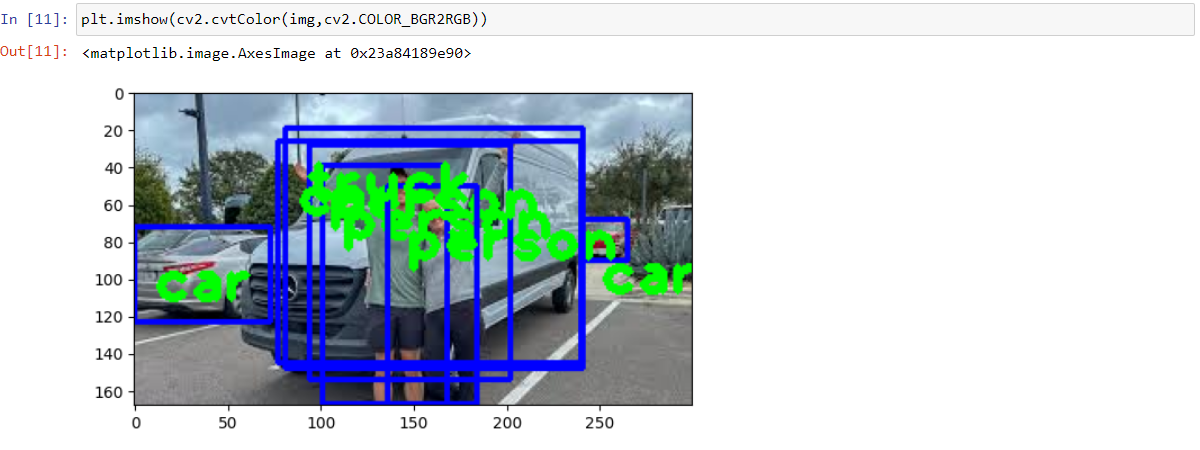
Then I draw bounding boxes and labels.

A computer screen shot of a computer screen

Description automatically generated

for class\_index, confidence, bounding\_boxes in zip(...) iterates over the flattened lists of class indices, confidence scores, and bounding box coordinates. cv2.rectangle(img, bounding\_boxes, (255, 0, 0), 2)draws a bounding box around the detected object with blue color and a thickness of 2. cv2.putText(...) labels the bounding boxes with the relevant class labels.

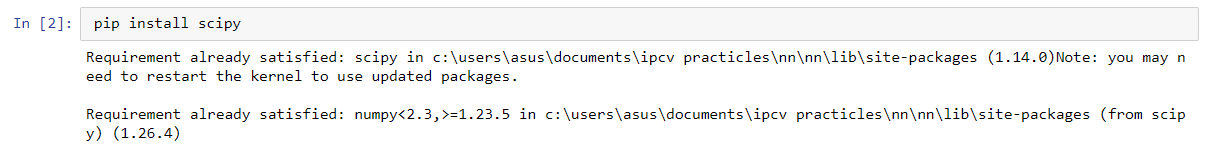
Finally display the final image with bounding boxes and labels, converting the image back to RGB format.



**Practical 06: Pneumonia Detection**

First, I ensure that TensorFlow and SciPy are installed:







I start by importing the necessary libraries for building and training the model:A screenshot of a computer program

Description automatically generated**Loading and Modifying VGG16 Model**

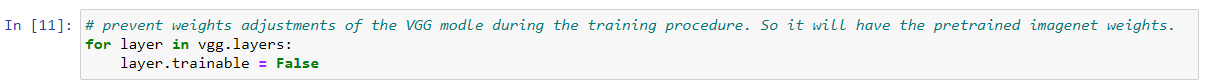
Next, I load the VGG16 model without the top fully connected layers. This allows me to use the pre-trained model's convolutional base:

A white box with text

Description automatically generated

I discard the final fully connected layer because I need to customize the final outputs. VGG16 originally has around 1000 output classes, but for pneumonia prediction, I only need two classes: positive and negative.

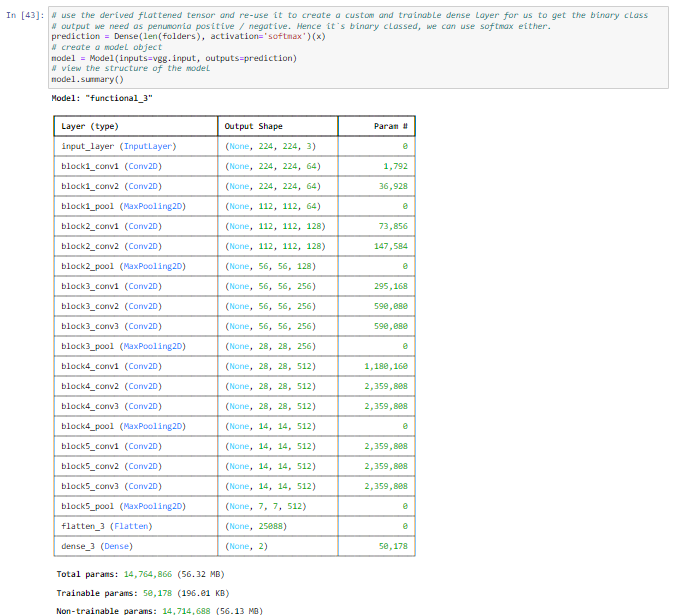
To prevent the weights of the VGG16 model from being adjusted during training, I set all layers to be non-trainable:



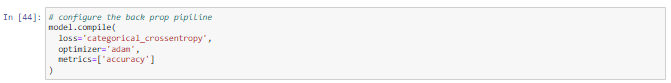
Then I get the number of output classes (in this case, folders in the training set) and flatten the output of the VGG16 model.



Again, I add a dense layer with softmax activation to get the binary class output (pneumonia positive/negative) and create the final model by specifying the inputs and outputs.



Next, I configure the backpropagation pipeline by compiling the model.



I use ImageDataGenerator for data augmentation to generate more diverse training images. For the test set, I only rescaled the images. Then I create the training set generator and similarly, I create the test set generator.

A screenshot of a computer code

Description automatically generated

Then I train the model using the training and test sets. I set the number of epochs to 1 for demonstration purposes (in practice, this would be higher)

A screenshot of a computer

Description automatically generated

After training save the model to a file.

A screen shot of a computer

Description automatically generated

Then I load the model.

A screenshot of a computer error

Description automatically generated

Next, I load a new image and preprocess it for prediction and use the model to predict the class of the images.

A close-up of a computer screen

Description automatically generated

After that print the prediction result.



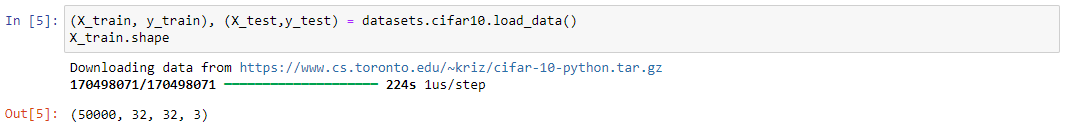
**Practical 07: Multiclass Image Classification**

Firstly, I import relevant libraries.



* numpy: A fundamental package for numerical computing in Python.
* tensorflow.keras: TensorFlow's high-level API for building and training deep learning models.
* matplotlib.pyplot: A plotting library for creating static, animated, and interactive visualizations in Python.
* keras.datasets: Provides built-in datasets like CIFAR-10.
* keras.layers: Contains layers used to build neural networks.
* keras.models: Provides the interface to define and train models.

Doing this practicle I use CIFAR-10 dataset. CIFAR-10 is a dataset of 60,000 32x32 color images in 10 classes, with 6,000 images per class. It is divided into 50,000 training images and 10,000 test images.



datasets.cifar10.load\_data() command loads the CIFAR-10 dataset and(X\_train, y\_train), (X\_test, y\_test)unpacks the training and testing data.

A white screen with blue text

Description automatically generated

Then I inspect the shapes of the arrays. Prints the shapes of X\_train, X\_test, y\_train, and y\_test. X\_train and X\_test are 4D arrays (number of images, height, width, channels). y\_train and y\_test are 2D arrays (number of images, 1).

Then I reshape the y\_train and y\_test and prints the shapes after reshaping.

A white card with black text

Description automatically generated

Then I need to view the images from the dataset.

A white screen with blue text

Description automatically generated

Then I create a function to label and display images.

A white background with black text

Description automatically generated

label\_and\_display\_image(X, y, index) defines a function to display an image from the dataset with its label. plt.figure(figsize=(15, 2)) creates a new figure for plotting with specified size and plt.imshow(X[index]) displays the image at the specified index. plt.xlabel(classes[y[index]])labels the image using the class name from the classes list.

Finally, I can view the images.

A screenshot of a computer

Description automatically generated

label\_and\_display\_image(X\_train, y\_train, 12225) displays and labels the image at index 12225 from the training set.

More Images

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated