ImageAI Documentation

Release 2.0.2

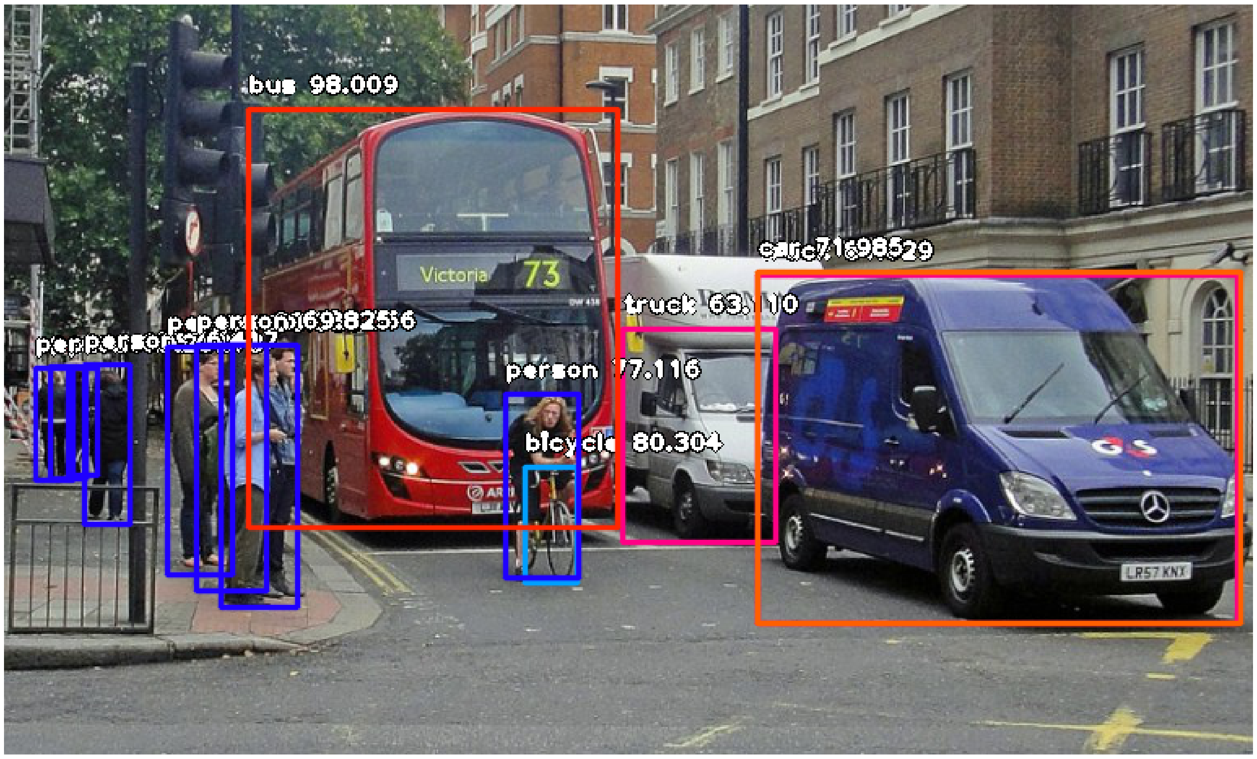
"Moses Olafenwa" "John Olafenwa"

Oct 28, 2018

Contents:

ImageAI is a python library built to empower developers, reseachers and students to build applications and systems with self-contained Deep Learning and Computer Vision capabilities using simple and few lines of code. This documentation is provided to provide detailed insight into all the classes and functions available in **ImageAI**, coupled with a number of code examples. **ImageAI** is a project developed by Moses Olafenwa and John Olafenwa , the DeepQuest AI team.

The Official GitHub Repository of **ImageAI** is https://github.com/OlafenwaMoses/ImageAI



CHAPTER 1

Installing ImageAI

ImageAI requires that you have Python 3.5.1 or higher installed as well as some other Python libraries and frameworks. Before you install **ImageAI**, you must install the following dependencies.

- **Python** 3.5.1 or higher, Download Python
- **pip3** , Download PyPi
- **Tensorflow** 1.4.0 or higher

  ```
  pip3 install --upgrade tensorflow
  ```

- **Numpy** 1.13.1 or higher

  ```
  pip3 install numpy
  ```

- **SciPy** .19.1 or higher

  ```
  pip3 install scipy
  ```

- **OpenCV**

  ```
  pip3 install opencv-python
  ```

- **Pillow**

  ```
  pip3 install pillow
  ```

- **Matplotlib**

  ```
  pip3 install matplotlib
  ```

- **h5py**

  ```
  pip3 install h5py
  ```

- **Keras**

```
pip3 install keras
```

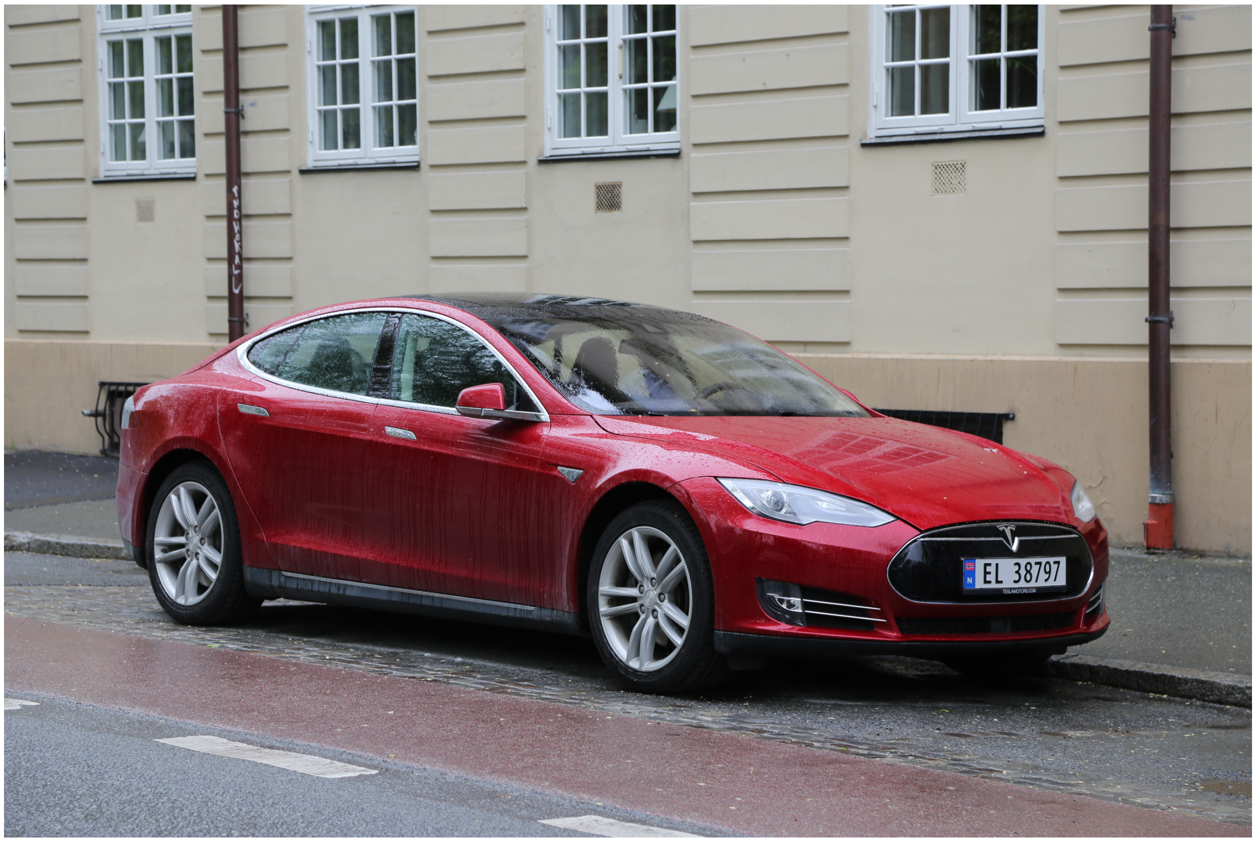
Once you have these packages installed on your computer system, you should install **ImageAI** by running the pip command below. Installing **ImageAI**

```
pip3 install https://github.com/OlafenwaMoses/ImageAI/releases/download/2.0.2/imageai-
↪2.0.2-py3-none-any.whl
```

Once **ImageAI** is installed, you can start running very few lines of code to perform very powerful computer visions tasks as seen below.

Image Recognition

Find all sample codes and documentation via links in the Content secton below this page.



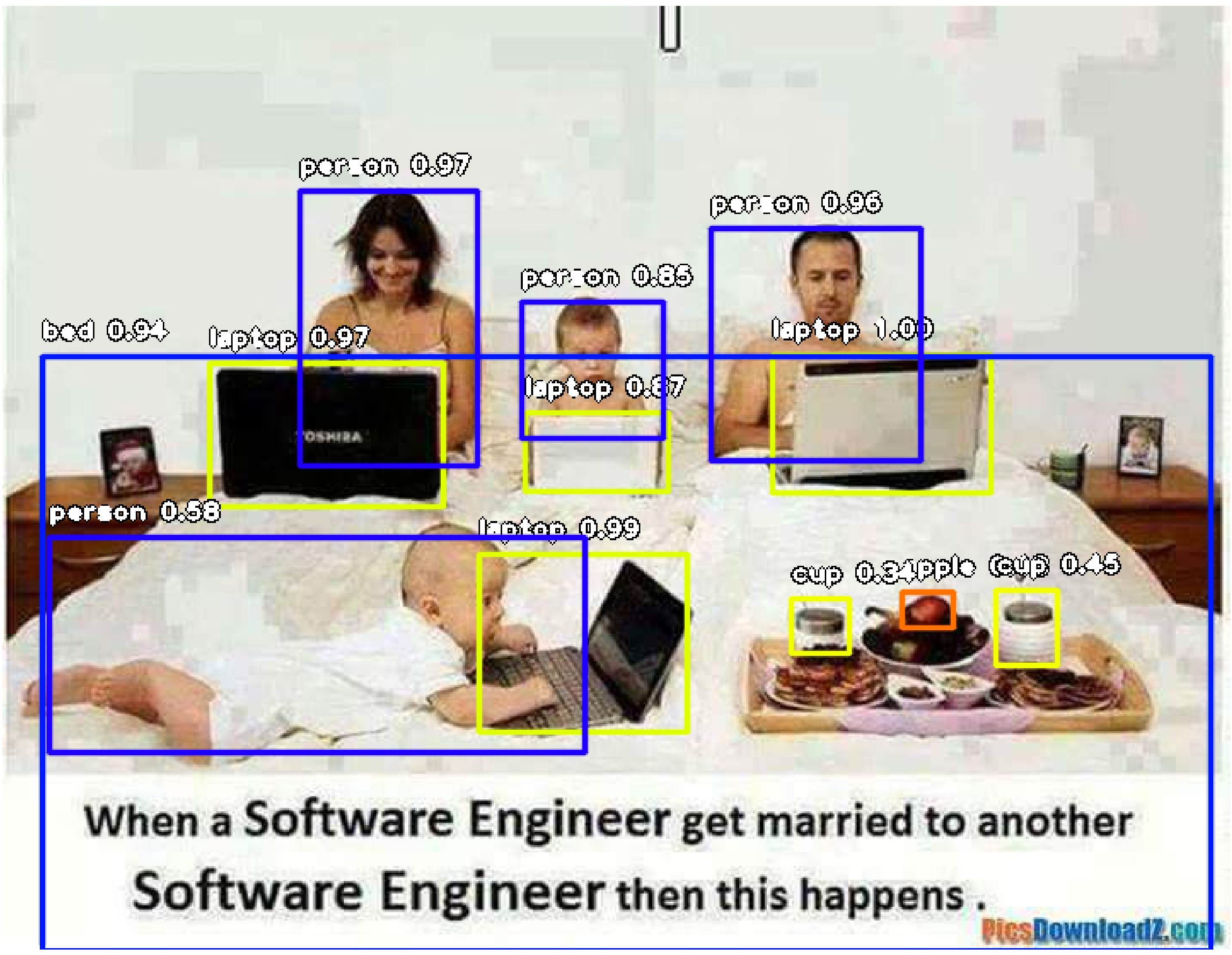
- convertible : 52.459555864334106
- sports_car : 37.61284649372101
- pickup : 3.1751200556755066
- car_wheel : 1.817505806684494
- minivan : 1.7487050965428352

** Image Object Detection**

Find all sample codes and documentation via links in the Content secton below this page.

Video Object Detection

Find all sample codes and documentation via links in the Content secton below this page.


person 0.97
person 0.85
person 0.96
bed 0.94
laptop 0.97
laptop 0.87
laptop 1.00
person 0.58
laptop 0.99
cup 0.34
apple
cup 0.45
When a Software Engineer get married to another Software Engineer then this happens .
PicsDownloadZ.com

Video Detection Analysis

Find all sample codes and documentation via links in the Content secton below this page.

Custom Image Recognition Training and Inference

Find all sample codes and documentation via links in the Content secton below this page.

Follow the links in the **Content** section below to see all the code samples and full documentation of available classes and functions.

1.1 Prediction Classes

ImageAI provides very powerful yet easy to use classes to perform **Image Recognition** tasks. You can perform all of these state-of-the-art computer vision tasks with python code that ranges between just 5 lines to 12 lines. Once you have Python, other dependencies and **ImageAI** installed on your computer system, there is no limit to the incredible applications you can create. Find below the classes and their respective functions available for you to use. These classes can be integrated into any traditional python program you are developing, be it a website, Windows/Linux/MacOS application or a system that supports or part of a Local-Area-Network.

======= imageai.Prediction.ImagePrediction =======

The **ImagePrediction** class provides you the functions to use state-of-the-art image recognition models like **SqueezeNet**, **ResNet**, **InceptionV3** and **DenseNet** that were **pre-trained** on the the **ImageNet-1000** dataset.This means you can use this class to predict/recognize 1000 different objects in any image or number of images. To initiate the class in your code, you will create a new instance of the class in your code as seen below

```
from imageai.Prediction import ImagePrediction
prediction = ImagePrediction()
```

We have provided pre-trained **SqueezeNet**, **ResNet**, **InceptionV3** and **DenseNet** image recognition models which you use with your **ImagePrediction** class to recognize images. Find below the link to download the pre-trained models. You can download the model you want to use.

Download SqueezeNet Model

Download ResNet Model

Download InceptionV3 Model

Download DenseNet Model

After creating a new instance of the **ImagePrediction** class, you can use the functions below to set your instance property and start recognizing objects in images.

- **.setModelTypeAsSqueezeNet()** , This function sets the model type of the image recognition instance you created to the **SqueezeNet** model, which means you will be performing your image prediction tasks using the pre-trained "SqueezeNet" model you downloaded from the links above. Find example code below

  ```
  prediction.setModelTypeAsSqueezeNet()
  ```

- **.setModelTypeAsResNet()** , This function sets the model type of the image recognition instance you created to the **ResNet** model, which means you will be performing your image prediction tasks using the pre-trained "ResNet" model you downloaded from the links above. Find example code below

  ```
  prediction.setModelTypeAsResNet()
  ```

- **.setModelTypeAsInceptionV3()** , This function sets the model type of the image recognition instance you created to the **InecptionV3** model, which means you will be performing your image prediction tasks using the pre-trained "InceptionV3" model you downloaded from the links above. Find example code below

  ```
  prediction.setModelTypeAsInceptionV3()
  ```

- **.setModelTypeAsDenseNet()** , This function sets the model type of the image recognition instance you created to the **DenseNet** model, which means you will be performing your image prediction tasks using the pre-trained "DenseNet" model you downloaded from the links above. Find example code below

  ```
  prediction.setModelTypeAsDenseNet()
  ```

- **.setModelPath()** , This function accepts a string which must be the path to the model file you downloaded and must corresponds to the model type you set for your image prediction instance. Find example code,and parameters of the function below

  ```
  prediction.setModelPath("resnet50_weights_tf_dim_ordering_tf_kernels.h5")
  ```

  – *parameter* **model_path** (required) : This is the path to your downloaded model file.

- **.loadModel()** , This function loads the model from the path you specified in the function call above into your image prediction instance. Find example code below

  ```
  prediction.loadModel()
  ```

– *parameter* **prediction_speed** (optional) : This parameter allows you to reduce the time it takes to predict in an image by up to 80% which leads to slight reduction in accuracy. This parameter accepts string values. The available values are "normal", "fast", "faster" and "fastest". The default values is "normal"

- **.predictImage()** , This is the function that performs actual prediction of an image. It can be called many times on many images once the model as been loaded into your prediction instance. Find example code,parameters of the function and returned values below

```
predictions, probabilities = prediction.predictImage("image1.jpg", result_
↪count=10)
```

– *parameter* **image_input** (required) : This refers to the path to your image file, Numpy array of your image or image file stream of your image, depending on the input type you specified.

—*parameter* **result_count** (optional) : This refers to the number of possible predictions that should be returned. The parameter is set to 5 by default.

– *parameter* **input_type** (optional) : This refers to the type of input you are parse into the **image_input** parameter. It is "file" by default and it accepts "array" and "stream" as well.

—*returns* **prediction_results** (a python list) : The first value returned by the **predictImage** function is a list that contains all the possible prediction results. The results are arranged in descending order of the percentage probability.

– *returns* **prediction_probabilities** (a python list) : The second value returned by the **predictImage** function is a list that contains the corresponding percentage probability of all the possible predictions in the **prediction_results**.

- **.predictMultipleImages()** , This function can be used to perform prediction on 2 or more images at once. Find example code,parameters of the function and returned values below

```
results_array = multiple_prediction.predictMultipleImages(all_images_array,␣
↪result_count_per_image=5)

for each_result in results_array:
    predictions, percentage_probabilities = each_result["predictions"], each_
↪result["percentage_probabilities"]
    for index in range(len(predictions)):
        print(predictions[index] , " : " , percentage_probabilities[index])
    print("-----------------------")
```

– *parameter* **sent_images_array** (required) : This refers to a list that contains the path to your image files, Numpy array of your images or image file stream of your images, depending on the input type you specified.

– *parameter* **result_count_per_image** (optional) : This refers to the number of possible predictions that should be returned for each of the images. The parameter is set to 2 by default.

– *parameter* **input_type** (optional) : This refers to the format in which your images are in the list you parsed into the **sent_images_array** parameter. It is "file" by default and it accepts "array" and "stream" as well.

– *returns* **output_array** (a python list) : The value returned by the **predictMultipleImages** function is a list that contains dictionaries. Each dictionary correspondes the images contained in the array you parsed into the **sent_images_array**. Each dictionary has "prediction_results" property which is a list of athe prediction result for the image in that index as well as the "prediction_probabilities" which is a list of the corresponding percentage probability for each result.

Sample Codes

Find below sample code for predicting one image

```
from imageai.Prediction import ImagePrediction
import os

execution_path = os.getcwd()

prediction = ImagePrediction()
prediction.setModelTypeAsResNet()
prediction.setModelPath(os.path.join(execution_path, "resnet50_weights_tf_dim_
↪ordering_tf_kernels.h5"))
prediction.loadModel()

predictions, probabilities = prediction.predictImage(os.path.join(execution_path,
↪"image1.jpg"), result_count=10)
for eachPrediction, eachProbability in zip(predictions, probabilities):
    print(eachPrediction , " : " , eachProbability)
```

Find below sample code for prediction multiple images

```
from imageai.Prediction import ImagePrediction
import os

execution_path = os.getcwd()

multiple_prediction = ImagePrediction()
multiple_prediction.setModelTypeAsResNet()
multiple_prediction.setModelPath(os.path.join(execution_path, "resnet50_weights_tf_
↪dim_ordering_tf_kernels.h5"))
multiple_prediction.loadModel()

all_images_array = []

all_files = os.listdir(execution_path)
for each_file in all_files:
    if(each_file.endswith(".jpg") or each_file.endswith(".png")):
        all_images_array.append(each_file)

results_array = multiple_prediction.predictMultipleImages(all_images_array, result_
↪count_per_image=5)

for each_result in results_array:
    predictions, percentage_probabilities = each_result["predictions"], each_result[
↪"percentage_probabilities"]
    for index in range(len(predictions)):
        print(predictions[index] , " : " , percentage_probabilities[index])
    print("-----------------------")
```

1.2 Detection Classes

ImageAI provided very powerful yet easy to use classes and functions to perform ** Image Object Detection and Extraction.**
ImageAI allows you to perform all of these with state-of-the-art deep learning algorithms like **RetinaNet**, **YOLOv3** and **TinyYOLOv3**. With **ImageAI** you can run detection tasks and analyse images.

Find below the classes and their respective functions available for you to use. These classes can be integrated into any traditional python program you are developing, be it a website, Windows/Linux/MacOS application or a system that supports or part of a Local-Area-Network.

======= imageai.Detection.ObjectDetection =======

This **ObjectDetection** class provides you function to perform object detection on any image or set of images, using **pre-trained** models that was trained on the **COCO** dataset. The models supported are **RetinaNet**, **YOLOv3** and **TinyYOLOv3**. This means you can detect and recognize 80 different kind of common everyday objects. To get started, download any of the pre-trained model that you want to use via the links below.

Download RetinaNet Model - resnet50_coco_best_v2.0.1.h5

Download YOLOv3 Model - yolo.h5

Download TinyYOLOv3 Model - yolo-tiny.h5

Once you have downloaded the model of your choice, you should create a new instance of the **ObjectDetection** class as seen in the sample below:

```
from imageai.Detection import ObjectDetection

detector = ObjectDetection()
```

Once you have created an instance of the class, you can use the functions below to set your instance property and start detecting objects in images.

- **.setModelTypeAsRetinaNet()** , This function sets the model type of the object detection instance you created to the **RetinaNet** model, which means you will be performing your object detection tasks using the pre-trained "RetinaNet" model you downloaded from the links above. Find example code below:

  ```
  detector.setModelTypeAsRetinaNet()
  ```

- **.setModelTypeAsYOLOv3()** , This function sets the model type of the object detection instance you created to the **YOLOv3** model, which means you will be performing your object detection tasks using the pre-trained "YOLOv3" model you downloaded from the links above. Find example code below:

  ```
  detector.setModelTypeAsYOLOv3()
  ```

- **.setModelTypeAsTinyYOLOv3()** , This function sets the model type of the object detection instance you created to the **TinyYOLOv3** model, which means you will be performing your object detection tasks using the pre-trained "TinyYOLOv3" model you downloaded from the links above. Find example code below:

  ```
  detector.setModelTypeAsTinyYOLOv3()
  ```

- **.setModelPath()** , This function accepts a string which must be the path to the model file you downloaded and must corresponds to the model type you set for your object detection instance. Find example code,and parameters of the function below:

  ```
  detector.setModelPath("yolo.h5")
  ```

  – *parameter* **model_path** (required) : This is the path to your downloaded model file.

- **.loadModel()** , This function loads the model from the path you specified in the function call above into your object detection instance. Find example code below:

  ```
  detector.loadModel()
  ```

  – *parameter* **detection_speed** (optional) : This parameter allows you to reduce the time it takes to detect objects in an image by up to 80% which leads to slight reduction in accuracy. This parameter accepts string values. The available values are "normal", "fast", "faster", "fastest" and "flash". The default values is "normal"

- **.detectObjectsFromImage()** , This is the function that performs object detection task after the model as loaded. It can be called many times to detect objects in any number of images. Find example code below:

```
detections = detector.detectObjectsFromImage(input_image="image.jpg", output_
↪image_path="imagenew.jpg", minimum_percentage_probability=30)
```

– *parameter* **input_image** (required) : This refers to the path to image file which you want to detect. You can set this parameter to the Numpy array of File stream of any image if you set the paramter **input_type** to "array" or "stream"

—*parameter* **output_image_path** (required only if **input_type** = "file") : This refers to the file path to which the detected image will be saved. It is required only if **input_type** = "file"

– *parameter* **minimum_percentage_probability** (optional) : This parameter is used to determine the integrity of the detection results. Lowering the value shows more objects while increasing the value ensures objects with the highest accuracy are detected. The default value is 50.

—*parameter* **output_type** (optional) : This parameter is used to set the format in which the detected image will be produced. The available values are "file" and "array". The default value is "file". If this parameter is set to "array", the function will return a Numpy array of the detected image. See sample below::

returned_image, detections = detector.detectObjectsFromImage(input_image="image.jpg", output_type="array", minimum_percentage_probability=30)

—*parameter* **display_percentage_probability** (optional) : This parameter can be used to hide the percentage probability of each object detected in the detected image if set to False. The default values is True.

– *parameter* **display_object_name** (optional) : This parameter can be used to hide the name of each object detected in the detected image if set to False. The default values is True.

—*parameter* **extract_detected_objects** (optional) : This parameter can be used to extract and save/return each object detected in an image as a seperate image. The default values is False.

– *returns* : The returned values will depend on the parameters parsed into the **detectObjectsFromImage()** function. See the comments and code below:

```
"""
    If all required parameters are set and 'output_image_path' is set to a␣
↪file path you want the detected image to be saved, the function will␣
↪return:

    1. an array of dictionaries, with each dictionary corresponding to the␣
↪objects
        detected in the image. Each dictionary contains the following␣
↪property:
            * name (string)
            * percentage_probability (float)
            * box_points (tuple of x1,y1,x2 and y2 coordinates)
"""
detections = detector.detectObjectsFromImage(input_image="image.jpg", output_
↪image_path="imagenew.jpg", minimum_percentage_probability=30)


"""
    If all required parameters are set and output_type = 'array' ,the␣
↪function will return
```

(continues on next page)

(continued from previous page)

```
    1. a numpy array of the detected image
    2. an array of dictionaries, with each dictionary corresponding to the␣
↪objects
        detected in the image. Each dictionary contains the following␣
↪property:
            * name (string)
            * percentage_probability (float)
            * box_points (tuple of x1,y1,x2 and y2 coordinates)
"""
returned_image, detections = detector.detectObjectsFromImage(input_image=
↪"image.jpg", output_type="array", minimum_percentage_probability=30)


"""
    If extract_detected_objects = True and 'output_image_path' is set to a␣
↪file path you want
        the detected image to be saved, the function will return:
        1. an array of dictionaries, with each dictionary corresponding to␣
↪the objects
            detected in the image. Each dictionary contains the following␣
↪property:
            * name (string)
            * percentage_probability (float)
            * box_points (tuple of x1,y1,x2 and y2 coordinates)
        2. an array of string paths to the image of each object extracted␣
↪from the image
"""
detections, extracted_objects = detector.detectObjectsFromImage(input_image=
↪"image.jpg", output_image_path="imagenew.jpg", extract_detected_
↪objects=True, minimum_percentage_probability=30)


"""
    If extract_detected_objects = True and output_type = 'array', the the␣
↪function will return:
        1. a numpy array of the detected image
        2. an array of dictionaries, with each dictionary corresponding to␣
↪the objects
            detected in the image. Each dictionary contains the following␣
↪property:
            * name (string)
            * percentage_probability (float)
            * box_points (tuple of x1,y1,x2 and y2 coordinates)
        3. an array of numpy arrays of each object detected in the image
"""
returned_image, detections, extracted_objects = detector.
↪detectObjectsFromImage(input_image="image.jpg", output_type="array",␣
↪extract_detected_objects=True, minimum_percentage_probability=30)
```

- **.CustomObjects()** , This function is used when you want to detect only a selected number of objects. It returns a dictionary of objects and their True or False values. To detect selected objects in an image, you will have to use the dictionary returned by the this function with the **detectCustomObjectsFromImage()** function. Find the details in the comment and code sample below:

```
"""
There are 80 possible objects that you can detect with the
```

(continues on next page)

(continued from previous page)

```
ObjectDetection class, and they are as seen below.

    person,   bicycle,   car,   motorcycle,   airplane,
    bus,   train,   truck,   boat,   traffic light,   fire hydrant,   stop_sign,
    parking meter,   bench,   bird,   cat,   dog,   horse,   sheep,   cow,  ␣
↪elephant,   bear,   zebra,
    giraffe,   backpack,   umbrella,   handbag,   tie,   suitcase,   frisbee,  ␣
↪skis,   snowboard,
    sports ball,   kite,   baseball bat,   baseball glove,   skateboard,  ␣
↪surfboard,   tennis racket,
    bottle,   wine glass,   cup,   fork,   knife,   spoon,   bowl,   banana,  ␣
↪apple,   sandwich,   orange,
    broccoli,   carrot,   hot dog,   pizza,   donot,   cake,   chair,   couch,  ␣
↪potted plant,   bed,
    dining table,   toilet,   tv,   laptop,   mouse,   remote,   keyboard,   cell␣
↪phone,   microwave,
    oven,   toaster,   sink,   refrigerator,   book,   clock,   vase,   scissors,␣
↪   teddy bear,   hair dryer,
    toothbrush.

To detect only some of the objects above, you will need to call the CustomObjects␣
↪function and set the name of the
object(s) yiu want to detect to through. The rest are False by default. In below␣
↪example, we detected only chose detect only person and dog.
"""
custom = detector.CustomObjects(person=True, dog=True)
```

- **.detectCustomObjectsFromImage()**, This function have all the parameters and returns all the values the **detectObjectsFromImage()** functions does but a slight difference. This function let detect only selected objects in an image. Unlike the normal **detectObjectsFromImage()** function, this needs an extra parameter which is "custom_object" which accepts the dictionary returned by the **CustomObjects()** function. In the sample below, we set the detection funtion to report only detections on persons and dogs:

```
custom = detector.CustomObjects(person=True, dog=True)

detections = detector.detectCustomObjectsFromImage( custom_objects=custom, input_
↪image=os.path.join(execution_path , "image3.jpg"), output_image_path=os.path.
↪join(execution_path , "image3new-custom.jpg"), minimum_percentage_
↪probability=30)
```

Sample Image Object Detection code

Find below a code sample for detecting objects in an image:

```
from imageai.Detection import ObjectDetection
import os

execution_path = os.getcwd()

detector = ObjectDetection()
detector.setModelTypeAsYOLOv3()
detector.setModelPath( os.path.join(execution_path , "yolo.h5"))
detector.loadModel()
detections = detector.detectObjectsFromImage(input_image=os.path.join(execution_path ,
↪ "image.jpg"), output_image_path=os.path.join(execution_path , "imagenew.jpg"),␣
↪minimum_percentage_probability=30)
```

(continues on next page)

(continued from previous page)

```
for eachObject in detections:
    print(eachObject["name"] , " : ", eachObject["percentage_probability"], " : ",␣
↪eachObject["box_points"] )
    print("--------------------------------")
```

1.3 Video and Live-Feed Detection and Analysis

ImageAI provided very powerful yet easy to use classes and functions to perform **Video Object Detection and Tracking** and **Video analysis**. **ImageAI** allows you to perform all of these with state-of-the-art deep learning algorithms like **RetinaNet**, **YOLOv3** and **TinyYOLOv3**. With **ImageAI** you can run detection tasks and analyse videos and live-video feeds from device cameras and IP cameras. Find below the classes and their respective functions available for you to use. These classes can be integrated into any traditional python program you are developing, be it a website, Windows/Linux/MacOS application or a system that supports or part of a Local-Area-Network.

======= imageai.Detection.VideoObjectDetection =======

This **VideoObjectDetection** class provides you function to detect objects in videos and live-feed from device cameras and IP cameras, using **pre-trained** models that was trained on the **COCO** dataset. The models supported are **RetinaNet**, **YOLOv3** and **TinyYOLOv3**. This means you can detect and recognize 80 different kind of common everyday objects in any video. To get started, download any of the pre-trained model that you want to use via the links below.

Download RetinaNet Model - resnet50_coco_best_v2.0.1.h5

Download YOLOv3 Model - yolo.h5

Download TinyYOLOv3 Model - yolo-tiny.h5

Once you have downloaded the model you chose to use, create an instance of the **VideoObjectDetection** as seen below:

```
from imageai.Detection import VideoObjectDetection

detector = VideoObjectDetection()
```

Once you have created an instance of the class, you can call the functions below to set its properties and detect objects in a video.

- **.setModelTypeAsRetinaNet()** , This function sets the model type of the object detection instance you created to the **RetinaNet** model, which means you will be performing your object detection tasks using the pre-trained "RetinaNet" model you downloaded from the links above. Find example code below:

  ```
  detector.setModelTypeAsRetinaNet()
  ```

- **.setModelTypeAsYOLOv3()** , This function sets the model type of the object detection instance you created to the **YOLOv3** model, which means you will be performing your object detection tasks using the pre-trained "YOLOv3" model you downloaded from the links above. Find example code below:

  ```
  detector.setModelTypeAsYOLOv3()
  ```

- **.setModelTypeAsTinyYOLOv3()** , This function sets the model type of the object detection instance you created to the **TinyYOLOv3** model, which means you will be performing your object detection tasks using the pre-trained "TinyYOLOv3" model you downloaded from the links above. Find example code below:

  ```
  detector.setModelTypeAsTinyYOLOv3()
  ```

- **.setModelPath()** , This function accepts a string which must be the path to the model file you downloaded and must corresponds to the model type you set for your object detection instance. Find example code,and parameters of the function below:

```
detector.setModelPath("yolo.h5")
```

– *parameter* **model_path** (required) : This is the path to your downloaded model file.

- **.loadModel()** , This function loads the model from the path you specified in the function call above into your object detection instance. Find example code below:

```
detector.loadModel()
```

– *parameter* **detection_speed** (optional) : This parameter allows you to reduce the time it takes to detect objects in a video by up to 80% which leads to slight reduction in accuracy. This parameter accepts string values. The available values are "normal", "fast", "faster", "fastest" and "flash". The default values is "normal"

- **.detectObjectsFromVideo()** , This is the function that performs object detecttion on a video file or video live-feed after the model has been loaded into the instance you created. Find a full sample code below:

```
from imageai.Detection import VideoObjectDetection
import os

execution_path = os.getcwd()

detector = VideoObjectDetection()
detector.setModelTypeAsYOLOv3()
detector.setModelPath( os.path.join(execution_path , "yolo.h5"))
detector.loadModel()

video_path = detector.detectObjectsFromVideo(input_file_path=os.path.
↪join(execution_path, "traffic.mp4"),
                                output_file_path=os.path.join(execution_path,
↪"traffic_detected")
                                , frames_per_second=20, log_progress=True)
print(video_path)
```

– *parameter* **input_file_path** (required if you did not set **camera_input**) : This refers to the path to the video file you want to detect.

—*parameter* **output_file_path** (required if you did not set **save_detected_video** = False) : This refers to the path to which the detected video will be saved. By default, this functionsaves video **.avi** format.

– *parameter* **frames_per_second** (optional , but recommended) : This parameters allows you to set your desired frames per second for the detected video that will be saved. The default value is 20 but we recommend you set the value that suits your video or camera live-feed.

—*parameter* **log_progress** (optional) : Setting this parameter to True shows the progress of the video or live-feed as it is detected in the CLI. It will report every frame detected as it progresses. The default value is False.

– *parameter* **return_detected_frame** (optional) : This parameter allows you to return the detected frame as a Numpy array at every frame, second and minute of the video detected. The returned Numpy array will be parsed into the respective **per_frame_function**, **per_second_function** and **per_minute_function** (See details below)

—*parameter* **camera_input** (optional) : This parameter can be set in replacement of the **input_file_path** if you want to detect objects in the live-feed of a camera. All you need is to load the camera with OpenCV's **VideoCapture()** function and parse the object into this parameter.

See a full code sample below:

```
from imageai.Detection import VideoObjectDetection
import os
import cv2

execution_path = os.getcwd()

camera = cv2.VideoCapture(0)

detector = VideoObjectDetection()
detector.setModelTypeAsYOLOv3()
detector.setModelPath(os.path.join(execution_path , "yolo.h5"))
detector.loadModel()

video_path = detector.detectObjectsFromVideo(camera_input=camera,
    output_file_path=os.path.join(execution_path, "camera_detected_
↪video")
    , frames_per_second=20, log_progress=True, minimum_percentage_
↪probability=30)

print(video_path)
```

—*parameter* **minimum_percentage_probability** (optional) : This parameter is used to determine the integrity of the detection results. Lowering the value shows more objects while increasing the value ensures objects with the highest accuracy are detected. The default value is 50.

– *parameter* **display_percentage_probability** (optional) : This parameter can be used to hide the percentage probability of each object detected in the detected video if set to False. The default values is True.

—*parameter* **display_object_name** (optional) : This parameter can be used to hide the name of each object detected in the detected video if set to False. The default values is True.

– *parameter* **save_detected_video** (optional) : This parameter can be used to or not to save the detected video or not to save it. It is set to True by default.

—*parameter* **per_frame_function** (optional) : This parameter allows you to parse in the name of a function you define. Then, for every frame of the video that is detected, the function will be parsed into the parameter will be executed and and analytical data of the video will be parsed into the function. The data returned can be visualized or saved in a NoSQL database for future processing and visualization.

See the sample code below:

```
"""
This parameter allows you to parse in a function you will want to␣
↪execute after
each frame of the video is detected. If this parameter is set to a␣
↪function, after every video
frame is detected, the function will be executed with the following␣
↪values parsed into it:
-- position number of the frame
-- an array of dictinaries, with each dictinary corresponding to␣
↪each object detected.
    Each dictionary contains 'name', 'percentage_probability' and
↪'box_points'
-- a dictionary with with keys being the name of each unique objects␣
↪and value
   are the number of instances of each of the objects present
```

(continues on next page)

(continued from previous page)

```
-- If return_detected_frame is set to True, the numpy array of the␣
↪detected frame will be parsed
    as the fourth value into the function
"""

from imageai.Detection import VideoObjectDetection
import os


def forFrame(frame_number, output_array, output_count):
print("FOR FRAME " , frame_number)
print("Output for each object : ", output_array)
print("Output count for unique objects : ", output_count)
print("------------END OF A FRAME --------------")


video_detector = VideoObjectDetection()
video_detector.setModelTypeAsYOLOv3()
video_detector.setModelPath(os.path.join(execution_path, "yolo.h5"))
video_detector.loadModel()


video_detector.detectObjectsFromVideo(input_file_path=os.path.
↪join(execution_path, "traffic.mp4"), output_file_path=os.path.
↪join(execution_path, "video_frame_analysis") ,  frames_per_
↪second=20, per_frame_function=forFrame,  minimum_percentage_
↪probability=30)
```

In the above example, once every frame in the video is processed and detected, the function will receive and prints out the analytical data for objects detected in the video frame as you can see below:

```
Output for each object : [{'box_points': (362, 295, 443, 355), 'name
↪': 'boat', 'percentage_probability': 26.666194200515747}, {'box_
↪points': (319, 245, 386, 296), 'name': 'boat', 'percentage_
↪probability': 30.052968859672546}, {'box_points': (219, 308, 341,␣
↪358), 'name': 'boat', 'percentage_probability': 47.46982455253601},
↪ {'box_points': (589, 198, 621, 241), 'name': 'bus', 'percentage_
↪probability': 24.62330162525177}, {'box_points': (519, 181, 583,␣
↪263), 'name': 'bus', 'percentage_probability': 27.446213364601135},
↪ {'box_points': (493, 197, 561, 272), 'name': 'bus', 'percentage_
↪probability': 59.81815457344055}, {'box_points': (432, 187, 491,␣
↪240), 'name': 'bus', 'percentage_probability': 64.42965269088745},
↪{'box_points': (157, 225, 220, 255), 'name': 'car', 'percentage_
↪probability': 21.150341629981995}, {'box_points': (324, 249, 377,␣
↪293), 'name': 'car', 'percentage_probability': 24.089913070201874},
↪ {'box_points': (152, 275, 260, 327), 'name': 'car', 'percentage_
↪probability': 30.341443419456482}, {'box_points': (433, 198, 485,␣
↪244), 'name': 'car', 'percentage_probability': 37.205660343170166},
↪ {'box_points': (184, 226, 233, 260), 'name': 'car', 'percentage_
↪probability': 38.52525353431702}, {'box_points': (3, 296, 134,␣
↪359), 'name': 'car', 'percentage_probability': 47.80363142490387},
↪{'box_points': (357, 302, 439, 359), 'name': 'car', 'percentage_
↪probability': 47.94844686985016}, {'box_points': (481, 266, 546,␣
↪314), 'name': 'car', 'percentage_probability': 65.8585786819458}, {
↪'box_points': (597, 269, 624, 318), 'name': 'person', 'percentage_
↪probability': 27.125394344329834}]
```

(continues on next page)

(continued from previous page)

```
Output count for unique objects : {'bus': 4, 'boat': 3, 'person': 1,
↪'car': 8}

------------END OF A FRAME --------------
```

Below is a full code that has a function that taskes the analyitical data and visualizes it and the detected frame in real time as the video is processed and detected:

```
from imageai.Detection import VideoObjectDetection
import os
from matplotlib import pyplot as plt


execution_path = os.getcwd()

color_index = {'bus': 'red', 'handbag': 'steelblue', 'giraffe':
↪'orange', 'spoon': 'gray', 'cup': 'yellow', 'chair': 'green',
↪'elephant': 'pink', 'truck': 'indigo', 'motorcycle': 'azure',
↪'refrigerator': 'gold', 'keyboard': 'violet', 'cow': 'magenta',
↪'mouse': 'crimson', 'sports ball': 'raspberry', 'horse': 'maroon',
↪'cat': 'orchid', 'boat': 'slateblue', 'hot dog': 'navy', 'apple':
↪'cobalt', 'parking meter': 'aliceblue', 'sandwich': 'skyblue',
↪'skis': 'deepskyblue', 'microwave': 'peacock', 'knife': 'cadetblue
↪', 'baseball bat': 'cyan', 'oven': 'lightcyan', 'carrot': 'coldgrey
↪', 'scissors': 'seagreen', 'sheep': 'deepgreen', 'toothbrush':
↪'cobaltgreen', 'fire hydrant': 'limegreen', 'remote': 'forestgreen
↪', 'bicycle': 'olivedrab', 'toilet': 'ivory', 'tv': 'khaki',
↪'skateboard': 'palegoldenrod', 'train': 'cornsilk', 'zebra': 'wheat
↪', 'tie': 'burlywood', 'orange': 'melon', 'bird': 'bisque',
↪'dining table': 'chocolate', 'hair drier': 'sandybrown', 'cell␣
↪phone': 'sienna', 'sink': 'coral', 'bench': 'salmon', 'bottle':
↪'brown', 'car': 'silver', 'bowl': 'maroon', 'tennis racket':
↪'palevilotered', 'airplane': 'lavenderblush', 'pizza': 'hotpink',
↪'umbrella': 'deeppink', 'bear': 'plum', 'fork': 'purple', 'laptop
↪': 'indigo', 'vase': 'mediumpurple', 'baseball glove': 'slateblue',
↪ 'traffic light': 'mediumblue', 'bed': 'navy', 'broccoli':
↪'royalblue', 'backpack': 'slategray', 'snowboard': 'skyblue', 'kite
↪': 'cadetblue', 'teddy bear': 'peacock', 'clock': 'lightcyan',
↪'wine glass': 'teal', 'frisbee': 'aquamarine', 'donut': 'mincream',
↪ 'suitcase': 'seagreen', 'dog': 'springgreen', 'banana':
↪'emeraldgreen', 'person': 'honeydew', 'surfboard': 'palegreen',
↪'cake': 'sapgreen', 'book': 'lawngreen', 'potted plant':
↪'greenyellow', 'toaster': 'ivory', 'stop sign': 'beige', 'couch':
↪'khaki'}


resized = False

def forFrame(frame_number, output_array, output_count, returned_
↪frame):

    plt.clf()

    this_colors = []
    labels = []
```

(continues on next page)

(continued from previous page)

```
    sizes = []

    counter = 0

    for eachItem in output_count:
        counter += 1
        labels.append(eachItem + " = " + str(output_count[eachItem]))
        sizes.append(output_count[eachItem])
        this_colors.append(color_index[eachItem])

    global resized

    if (resized == False):
        manager = plt.get_current_fig_manager()
        manager.resize(width=1000, height=500)
        resized = True

    plt.subplot(1, 2, 1)
    plt.title("Frame : " + str(frame_number))
    plt.axis("off")
    plt.imshow(returned_frame, interpolation="none")

    plt.subplot(1, 2, 2)
    plt.title("Analysis: " + str(frame_number))
    plt.pie(sizes, labels=labels, colors=this_colors, shadow=True,␣
↪startangle=140, autopct="%1.1f%%")

    plt.pause(0.01)



video_detector = VideoObjectDetection()
video_detector.setModelTypeAsYOLOv3()
video_detector.setModelPath(os.path.join(execution_path, "yolo.h5"))
video_detector.loadModel()

plt.show()

video_detector.detectObjectsFromVideo(input_file_path=os.path.
↪join(execution_path, "traffic.mp4"), output_file_path=os.path.
↪join(execution_path, "video_frame_analysis") ,  frames_per_
↪second=20, per_frame_function=forFrame,  minimum_percentage_
↪probability=30, return_detected_frame=True)
```

—*parameter* **per_second_function** (optional) : This parameter allows you to parse in the name of a function you define. Then, for every second of the video that is detected, the function will be parsed into the parameter will be executed and analytical data of the video will be parsed into the function. The data returned can be visualized or saved in a NoSQL database for future processing and visualization.

See the sample code below:

```
"""
This parameter allows you to parse in a function you will want to␣
↪execute after
each second of the video is detected. If this parameter is set to a␣
↪function, after every second of a video
is detected, the function will be executed with the following values␣
↪parsed into it:
```

(continues on next page)

(continued from previous page)

```
-- position number of the second
-- an array of dictionaries whose keys are position number of each␣
↪frame present in the last second , and the value for each key is␣
↪the array for each frame that contains the dictionaries for each␣
↪object detected in the frame

-- an array of dictionaries, with each dictionary corresponding to␣
↪each frame in the past second, and the keys of each dictionary are␣
↪the name of the number of unique objects detected in each frame,␣
↪and the key values are the number of instances of the objects␣
↪found in the frame

-- a dictionary with its keys being the name of each unique object␣
↪detected throughout the past second, and the key values are the␣
↪average number of instances of the object found in all the frames␣
↪contained in the past second

-- If return_detected_frame is set to True, the numpy array of the␣
↪detected frame will be parsed as the fifth value into the function
"""

from imageai.Detection import VideoObjectDetection
import os


def forSeconds(second_number, output_arrays, count_arrays, average_
↪output_count):
    print("SECOND : ", second_number)
    print("Array for the outputs of each frame ", output_arrays)
    print("Array for output count for unique objects in each frame :
↪", count_arrays)
    print("Output average count for unique objects in the last␣
↪second: ", average_output_count)
    print("------------END OF A SECOND --------------")


video_detector = VideoObjectDetection()
video_detector.setModelTypeAsYOLOv3()
video_detector.setModelPath(os.path.join(execution_path, "yolo.h5"))
video_detector.loadModel()


video_detector.detectObjectsFromVideo(input_file_path=os.path.
↪join(execution_path, "traffic.mp4"), output_file_path=os.path.
↪join(execution_path, "video_second_analysis") ,  frames_per_
↪second=20, per_second_function=forSecond,  minimum_percentage_
↪probability=30)
```

In the above example, once every second in the video is processed and detected, the function will receive and prints out the analytical data for objects detected in the video as you can see below:

```
Array for the outputs of each frame [[{'box_points': (362, 295, 443,␣
↪355), 'name': 'boat', 'percentage_probability': 26.666194200515747}
↪, {'box_points': (319, 245, 386, 296), 'name': 'boat', 'percentage_
↪probability': 30.052968859672546}, {'box_points': (219, 308, 341,␣
↪358), 'name': 'boat', 'percentage_probability': 47.46982455253601},
↪ {'box_points': (589, 198, 621, 241), 'name': 'bus', 'percentage_
↪probability': 24.62330162525177}, {'box_points': (519, 181, 583,␣
↪263), 'name': 'bus', 'percentage_probability': 27.446213364601135},
↪ {'box_points': (493, 197, 561, 272), 'name': 'bus', 'percentage_
↪probability': 59.81815457344055}, {'box_points': (432, 187, 491,␣
↪240), 'name': 'bus', 'percentage_probability': 64.42965269088745},
↪{'box_points': (157, 225, 220, 255), 'name': 'car', 'percentage_
```

(continues on next page)

(continued from previous page)

```
    [{'box_points': (316, 240, 384, 302), 'name': 'boat',
↪'percentage_probability': 29.594269394874573}, {'box_points': (361,
↪ 295, 441, 354), 'name': 'boat', 'percentage_probability': 36.
↪11513376235962}, {'box_points': (216, 305, 340, 357), 'name': 'boat
↪', 'percentage_probability': 44.89373862743378}, {'box_points':␣
↪(432, 198, 488, 244), 'name': 'truck', 'percentage_probability':␣
↪22.914741933345795}, {'box_points': (589, 199, 623, 240), 'name':
↪'bus', 'percentage_probability': 20.545457303524017}, {'box_points
↪': (519, 182, 583, 263), 'name': 'bus', 'percentage_probability':␣
↪24.467085301876068}, {'box_points': (492, 197, 563, 271), 'name':
↪'bus', 'percentage_probability': 61.112016439437866}, {'box_points
↪': (433, 188, 490, 241), 'name': 'bus', 'percentage_probability':␣
↪65.08989334106445}, {'box_points': (352, 303, 442, 357), 'name':
↪'car', 'percentage_probability': 20.025095343589783}, {'box_points
↪': (136, 172, 188, 195), 'name': 'car', 'percentage_probability':␣
↪21.571354568004608}, {'box_points': (152, 276, 261, 326), 'name':
↪'car', 'percentage_probability': 33.07966589927673}, {'box_points
↪': (181, 225, 230, 256), 'name': 'car', 'percentage_probability':␣
↪35.111838579177856}, {'box_points': (432, 198, 488, 244), 'name':
↪'car', 'percentage_probability': 36.25282347202301}, {'box_points
↪': (3, 292, 130, 360), 'name': 'car', 'percentage_probability': 67.
↪55480170249939}, {'box_points': (479, 265, 546, 314), 'name': 'car
↪', 'percentage_probability': 71.47912979125977}, {'box_points':␣
↪(597, 269, 625, 318), 'name': 'person', 'percentage_probability':␣
↪25.903674960136414}],................,
    [{'box_points': (133, 250, 187, 278), 'name': 'umbrella',
↪'percentage_probability': 21.518094837665558}, {'box_points': (154,
↪ 233, 218, 259), 'name': 'umbrella', 'percentage_probability': 23.
↪687003552913666}, {'box_points': (348, 311, 425, 360), 'name':
↪'boat', 'percentage_probability': 21.015766263008118}, {'box_points
↪': (11, 164, 137, 225), 'name': 'bus', 'percentage_probability':␣
↪32.20453858375549}, {'box_points': (424, 187, 485, 243), 'name':
↪'bus', 'percentage_probability': 38.043853640556335}, {'box_points
↪': (496, 186, 570, 264), 'name': 'bus', 'percentage_probability':␣
↪63.83994221687317}, {'box_points': (588, 197, 622, 240), 'name':
↪'car', 'percentage_probability': 23.51653128862381}, {'box_points
↪': (58, 268, 111, 303), 'name': 'car', 'percentage_probability':␣
↪24.538707733154297}, {'box_points': (2, 246, 72, 301), 'name': 'car
↪', 'percentage_probability': 28.433072566986084}, {'box_points':␣
↪(472, 273, 539, 323), 'name': 'car', 'percentage_probability': 87.
↪17672824859619}, {'box_points': (597, 270, 626, 317), 'name':
↪'person', 'percentage_probability': 27.459821105003357}]
    ]

Array for output count for unique objects in each frame : [{'bus': 4,
↪ 'boat': 3, 'person': 1, 'car': 8},
    {'truck': 1, 'bus': 4, 'boat': 3, 'person': 1, 'car': 7},
    {'bus': 5, 'boat': 2, 'person': 1, 'car': 5},
    {'bus': 5, 'boat': 1, 'person': 1, 'car': 9},
    {'truck': 1, 'bus': 2, 'car': 6, 'person': 1},
    {'truck': 2, 'bus': 4, 'boat': 2, 'person': 1, 'car': 7},
    {'truck': 1, 'bus': 3, 'car': 7, 'person': 1, 'umbrella': 1},
    {'bus': 4, 'car': 7, 'person': 1, 'umbrella': 2},
    {'bus': 3, 'car': 6, 'boat': 1, 'person': 1, 'umbrella': 3},
    {'bus': 3, 'car': 4, 'boat': 1, 'person': 1, 'umbrella': 2}]

Output average count for unique objects in the last second: {'truck
↪': 0.5, 'bus': 3.7, 'umbrella': 0.8, 'boat': 1.3, 'person': 1.0,
↪'car': 6.6}
```

(continues on next page)

(continued from previous page)

```

------------END OF A SECOND --------------
```

Below is a full code that has a function that taskes the analyitical data and visualizes it and the detected frame at the end of the second in real time as the video is processed and detected:

```
from imageai.Detection import VideoObjectDetection
import os
from matplotlib import pyplot as plt


execution_path = os.getcwd()

color_index = {'bus': 'red', 'handbag': 'steelblue', 'giraffe':
↪'orange', 'spoon': 'gray', 'cup': 'yellow', 'chair': 'green',
↪'elephant': 'pink', 'truck': 'indigo', 'motorcycle': 'azure',
↪'refrigerator': 'gold', 'keyboard': 'violet', 'cow': 'magenta',
↪'mouse': 'crimson', 'sports ball': 'raspberry', 'horse': 'maroon',
↪'cat': 'orchid', 'boat': 'slateblue', 'hot dog': 'navy', 'apple':
↪'cobalt', 'parking meter': 'aliceblue', 'sandwich': 'skyblue',
↪'skis': 'deepskyblue', 'microwave': 'peacock', 'knife': 'cadetblue
↪', 'baseball bat': 'cyan', 'oven': 'lightcyan', 'carrot': 'coldgrey
↪', 'scissors': 'seagreen', 'sheep': 'deepgreen', 'toothbrush':
↪'cobaltgreen', 'fire hydrant': 'limegreen', 'remote': 'forestgreen
↪', 'bicycle': 'olivedrab', 'toilet': 'ivory', 'tv': 'khaki',
↪'skateboard': 'palegoldenrod', 'train': 'cornsilk', 'zebra': 'wheat
↪', 'tie': 'burlywood', 'orange': 'melon', 'bird': 'bisque',
↪'dining table': 'chocolate', 'hair drier': 'sandybrown', 'cell␣
↪phone': 'sienna', 'sink': 'coral', 'bench': 'salmon', 'bottle':
↪'brown', 'car': 'silver', 'bowl': 'maroon', 'tennis racket':
↪'palevilotered', 'airplane': 'lavenderblush', 'pizza': 'hotpink',
↪'umbrella': 'deeppink', 'bear': 'plum', 'fork': 'purple', 'laptop
↪': 'indigo', 'vase': 'mediumpurple', 'baseball glove': 'slateblue',
↪ 'traffic light': 'mediumblue', 'bed': 'navy', 'broccoli':
↪'royalblue', 'backpack': 'slategray', 'snowboard': 'skyblue', 'kite
↪': 'cadetblue', 'teddy bear': 'peacock', 'clock': 'lightcyan',
↪'wine glass': 'teal', 'frisbee': 'aquamarine', 'donut': 'mincream',
↪ 'suitcase': 'seagreen', 'dog': 'springgreen', 'banana':
↪'emeraldgreen', 'person': 'honeydew', 'surfboard': 'palegreen',
↪'cake': 'sapgreen', 'book': 'lawngreen', 'potted plant':
↪'greenyellow', 'toaster': 'ivory', 'stop sign': 'beige', 'couch':
↪'khaki'}


resized = False

def forSecond(frame2_number, output_arrays, count_arrays, average_
↪count, returned_frame):

    plt.clf()

    this_colors = []
    labels = []
    sizes = []

    counter = 0
```

(continues on next page)

(continued from previous page)

```
    for eachItem in average_count:
        counter += 1
        labels.append(eachItem + " = " + str(average_
↪count[eachItem]))
        sizes.append(average_count[eachItem])
        this_colors.append(color_index[eachItem])

    global resized

    if (resized == False):
        manager = plt.get_current_fig_manager()
        manager.resize(width=1000, height=500)
        resized = True

    plt.subplot(1, 2, 1)
    plt.title("Second : " + str(frame_number))
    plt.axis("off")
    plt.imshow(returned_frame, interpolation="none")

    plt.subplot(1, 2, 2)
    plt.title("Analysis: " + str(frame_number))
    plt.pie(sizes, labels=labels, colors=this_colors, shadow=True,␣
↪startangle=140, autopct="%1.1f%%")

    plt.pause(0.01)



video_detector = VideoObjectDetection()
video_detector.setModelTypeAsYOLOv3()
video_detector.setModelPath(os.path.join(execution_path, "yolo.h5"))
video_detector.loadModel()

plt.show()

video_detector.detectObjectsFromVideo(input_file_path=os.path.
↪join(execution_path, "traffic.mp4"), output_file_path=os.path.
↪join(execution_path, "video_second_analysis") ,  frames_per_
↪second=20, per_second_function=forSecond,  minimum_percentage_
↪probability=30, return_detected_frame=True, log_progress=True)
```

—*parameter* **per_minute_function** (optional) : This parameter allows you to parse in the name of a function you define. Then, for every frame of the video that is detected, the function which was parsed into the parameter will be executed and analytical data of the video will be parsed into the function. The data returned has the same nature as the **per_second_function** ; the difference is that it covers all the frames in the past 1 minute of the video.

See a sample funtion for this parameter below:

```
def forMinute(minute_number, output_arrays, count_arrays, average_
↪output_count):
    print("MINUTE : ", minute_number)
    print("Array for the outputs of each frame ", output_arrays)
    print("Array for output count for unique objects in each frame :
↪", count_arrays)
    print("Output average count for unique objects in the last␣
↪minute: ", average_output_count)
```

(continues on next page)

(continued from previous page)

```
    print("------------END OF A MINUTE --------------")
```

—*parameter* **video_complete_function** (optional) : This parameter allows you to parse in the name of a function you define. Once all the frames in the video is fully detected, the function will was parsed into the parameter will be executed and analytical data of the video will be parsed into the function. The data returned has the same nature as the **per_second_function** and **per_minute_function** ; the differences are that no index will be returned and it covers all the frames in the entire video.

See a sample funtion for this parameter below:

```
def forFull(output_arrays, count_arrays, average_output_count):
    print("Array for the outputs of each frame ", output_arrays)
    print("Array for output count for unique objects in each frame :
↪", count_arrays)
    print("Output average count for unique objects in the entire␣
↪video: ", average_output_count)
    print("------------END OF THE VIDEO --------------")
```

1.4 Custom Training and Prediction Classes

ImageAI provides very powerful yet easy to use classes to train state-of-the-art deep learning algorithms like **SqueezeNet** , **ResNet** , **InceptionV3** and **DenseNet** on your own image datasets using as few as **5 lines of code** to generate your own custom models . Once you have trained your own custom model, you can use the **CustomImagePrediction** class provided by **ImageAI** to use your own models to recognize/predict any image or set of images.

======= imageai.Prediction.Custom.ModelTraining =======

The **ModelTraining** class allows you to train any of the 4 supported deep learning algorithms (**SqueezeNet** , **ResNet** , **InceptionV3** and **DenseNet**) on your own image dataset to generate your own custom models. Your image dataset must contain at least 2 different classes/types of images (e.g cat and dog) and you must collect at least 500 images for each of the classes to achieve maximum accuracy.

The training process generates a JSON file that maps the objects types in your image dataset and creates lots of models. You will then peak the model with the highest accuracy and perform custom image prediction using the model and the JSON file generated.

Because model training is a compute intensive tasks, we strongly advise you perform this experiment using a computer with a NVIDIA GPU and the GPU version of Tensorflow installed. Performing model training on CPU will my take hours or days. With NVIDIA GPU powered computer system, this will take a few hours. You can use Google Colab for this experiment as it has an NVIDIA K80 GPU available. To train a custom prediction model, you need to prepare the images you want to use to train the model. You will prepare the images as follows:

– Create a dataset folder with the name you will like your dataset to be called (e.g pets)

—In the dataset folder, create a folder by the name train

– In the dataset folder, create a folder by the name test

—In the train folder, create a folder for each object you want to the model to predict and give the folder a name that corresponds to the respective object name (e.g dog, cat, squirrel, snake)

– In the test folder, create a folder for each object you want to the model to predict and give the folder a name that corresponds to the respective object name (e.g dog, cat, squirrel, snake)

—In each folder present in the train folder, put the images of each object in its respective folder. This images are the ones to be used to train the model

– To produce a model that can perform well in practical applications, I recommend you about 500 or more images per object. 1000 images per object is just great

—In each folder present in the test folder, put about 100 to 200 images of each object in its respective folder. These images are the ones to be used to test the model as it trains

– Once you have done this, the structure of your image dataset folder should look like below

```
pets//train//dog//dog-train-images
pets//train//cat//cat-train-images
pets//train//squirrel//squirrel-train-images
pets//train//snake//snake-train-images

pets//test//dog//dog-test-images
pets//test//cat//cat-test-images
pets//test//squirrel//squirrel-test-images
pets//test//snake//snake-test-images
```

Once your dataset is ready, you can proceed to creating an instance of the **ModelTraining** class. Find the example below

```
from imageai.Prediction.Custom import ModelTraining

model_trainer = ModelTraining()
```

Once you have created an instance above, you can use the functions below to set your instance property and start the traning process.

- **.setModelTypeAsSqueezeNet()** , This function sets the model type of the training instance you created to the **SqueezeNet** model, which means the **SqueezeNet** algorithm will be trained on your dataset. Find example code below

```
model_trainer.setModelTypeAsSqueezeNet()
```

- **.setModelTypeAsResNet()** , This function sets the model type of the training instance you created to the **ResNet** model, which means the **ResNet** algorithm will be trained on your dataset. Find example code below

```
model_trainer.setModelTypeAsResNet()
```

- **.setModelTypeAsInceptionV3()** , This function sets the model type of the training instance you created to the **InceptionV3** model, which means the **InceptionV3** algorithm will be trained on your dataset. Find example code below

```
model_trainer.setModelTypeAsInceptionV3()
```

- **.setModelTypeAsDenseNet()** , This function sets the model type of the training instance you created to the **DenseNet** model, which means the **DenseNet** algorithm will be trained on your dataset. Find example code below

```
model_trainer.setModelTypeAsDenseNet()
```

- **.setDataDirectory()** , This function accepts a string which must be the path to the folder that contains the **test** and **train** subfolder of your image dataset. Find example code,and parameters of the function below

```
prediction.setDataDirectory(r"C:/Users/Moses/Documents/Moses/AI/Custom Datasets/
→pets")
```

– *parameter* **data_directory** (required) : This is the path to the folder that comtaims your image dataset.

- **.trainModel()** , This is the function that starts the training process. Once it starts, it will create a JSON file in the **dataset/json** folder (e.g **pets/json**) which contains the mapping of the classes of the dataset. The JSON file will be used during custom prediction to produce reults. Find exmaple code below

```
model_trainer.trainModel(num_objects=4, num_experiments=100, enhance_data=True,␣
↪batch_size=32, show_network_summary=True)
```

– *parameter* **num_objects** (required) : This refers to the number of different classes in your image dataset.

—*parameter* **num_experiments** (required) : This is the number of times the algorithm will be trained on your image dataset. The accuracy of your training does increases as the number of times it trains increases. However, it does peak after a certain number of trainings;and that point depends on the size and nature of the dataset.

– *parameter* **enhance_data** (optional) : This parameter is used to tranform your image dataset in order to generate more sample for training. It is set to False by default. However, it is useful to set it to True if your image dataset contains less than 1000 images per class.

—*parameter* **batch_size** (optional) : During training, the algorithm is trained on a set of images in parallel. Because of this, the default value is set to 32. You can increase or reduce this value if you understand well enough to know the capacity of the system you are using to train. Should you intend to chamge this value, you should set it to values that are in multiples of 8 to optimize the training process.

– *parameter* **show_network_summary** (optional) : This paramter when set to True displays the structure of the algorithm you are training on your image dataset in the CLI before training starts. It is set to False by default.

—*parameter* **initial_learning_rate** (optional) : This parameter is a highly technical value. It determines and control the behaviour of your training which is critical to the accuracy that can be achieved. Change this parameter's value only if you understand its function fully.

– *parameter* **initial_learning_rate** (optional) : This parameter is a highly technical value. It determines and control the behaviour of your training which is critical to the accuracy that can be achieved. Change this parameter's value only if you understand its function fully.

—*training_image_size* **initial_learning_rate** (optional) : This is the size at which the images in your image dataset will be trained, irrespective of their original sizes. The default value is 224 and must not be set to less than 100. Increasing this value increases accuracy but increases training time, and vice-versa.

Sample Code for Custom Model Training

Find below a sample code for training custom models for your image dataset

```
from imageai.Prediction.Custom import ModelTraining

model_trainer = ModelTraining()
model_trainer.setModelTypeAsResNet()
model_trainer.setDataDirectory(r"C:/Users/Moses/Documents/Moses/AI/Custom Datasets/
↪pets")
model_trainer.trainModel(num_objects=10, num_experiments=100, enhance_data=True,␣
↪batch_size=32, show_network_summary=True)
```

Below is a sample of the result when the training starts

```
Epoch 1/100
1/25 [>.............................] - ETA: 52s - loss: 2.3026 - acc: 0.2500
2/25 [=>............................] - ETA: 41s - loss: 2.3027 - acc: 0.1250
3/25 [==>...........................] - ETA: 37s - loss: 2.2961 - acc: 0.1667
4/25 [===>..........................] - ETA: 36s - loss: 2.2980 - acc: 0.1250
5/25 [=====>........................] - ETA: 33s - loss: 2.3178 - acc: 0.1000
```

(continues on next page)

(continued from previous page)

```
6/25 [======>......................] - ETA: 31s - loss: 2.3214 - acc: 0.0833
7/25 [=======>.....................] - ETA: 30s - loss: 2.3202 - acc: 0.0714
8/25 [========>....................] - ETA: 29s - loss: 2.3207 - acc: 0.0625
9/25 [=========>...................] - ETA: 27s - loss: 2.3191 - acc: 0.0556
10/25 [===========>..................] - ETA: 25s - loss: 2.3167 - acc: 0.0750
11/25 [============>.................] - ETA: 23s - loss: 2.3162 - acc: 0.0682
12/25 [=============>................] - ETA: 21s - loss: 2.3143 - acc: 0.0833
13/25 [==============>...............] - ETA: 20s - loss: 2.3135 - acc: 0.0769
14/25 [===============>..............] - ETA: 18s - loss: 2.3132 - acc: 0.0714
15/25 [=================>............] - ETA: 16s - loss: 2.3128 - acc: 0.0667
16/25 [==================>...........] - ETA: 15s - loss: 2.3121 - acc: 0.0781
17/25 [===================>..........] - ETA: 13s - loss: 2.3116 - acc: 0.0735
18/25 [====================>.........] - ETA: 12s - loss: 2.3114 - acc: 0.0694
19/25 [=====================>........] - ETA: 10s - loss: 2.3112 - acc: 0.0658
20/25 [=======================>......] - ETA: 8s - loss: 2.3109 - acc: 0.0625
21/25 [========================>.....] - ETA: 7s - loss: 2.3107 - acc: 0.0595
22/25 [=========================>....] - ETA: 5s - loss: 2.3104 - acc: 0.0568
23/25 [==========================>...] - ETA: 3s - loss: 2.3101 - acc: 0.0543
24/25 [===========================>..] - ETA: 1s - loss: 2.3097 - acc: 0.0625Epoch␣
↪00000: saving model to C:\Users\Moses\Documents\Moses\W7\AI\Custom␣
↪Datasets\IDENPROF\idenprof-small-test\idenprof\models\model_ex-000_acc-0.100000.h5

25/25 [==============================] - 51s - loss: 2.3095 - acc: 0.0600 - val_loss:␣
↪2.3026 - val_acc: 0.1000
```

Let us explain the details shown above:

1. The line Epoch 1/100 means the network is training the first experiment of the targeted 100
2. The line 1/25 [>…………………………..] - ETA: 52s - loss: 2.3026 - acc: 0.2500 represents the number of batches that has been trained in the present experiment
3. The line Epoch 00000: saving model to C:UsersUserPycharmProjectsImageAITestpetsmodelsmodelex-000acc-0.100000.h5 refers to the model saved after the present experiment. The ex_000 represents the experiment at this stage while the acc0.100000 and valacc: 0.1000 represents the accuracy of the model on the test images after the present experiment (maximum value value of accuracy is 1.0). This result helps to know the best performed model you can use for custom image prediction.

Once you are done training your custom model, you can use the **CustomImagePrediction** class described below to perform image prediction with your model.

======= imageai.Prediction.Custom.CustomImagePrediction =======

This class can be considered a replica of the **imageai.Prediction.ImagePrediction** as it has all the same functions, parameters and results. The only differences are that this class works with your own trained model, you will need to specify the path to the JSON file generated during the training and will need to specify the number of classes in your image dataset when loading the model. Below is an example of creating an instance of the class

```
from imageai.Prediction.Custom import CustomImagePrediction

prediction = CustomImagePrediction()
```

Once you have created the new instance, you can use the functions below to set your instance property and start recognizing objects in images.

- **.setModelTypeAsSqueezeNet()** , This function sets the model type of the image recognition instance you created to the **SqueezeNet** model, which means you will be performing your image prediction tasks using the “SqueezeNet” model generated during your custom training. Find example code below

```
prediction.setModelTypeAsSqueezeNet()
```

- **.setModelTypeAsResNet()** , This function sets the model type of the image recognition instance you created to the **ResNet** model, which means you will be performing your image prediction tasks using the "ResNet" model model generated during your custom training. Find example code below

```
prediction.setModelTypeAsResNet()
```

- **.setModelTypeAsInceptionV3()** , This function sets the model type of the image recognition instance you created to the **InecptionV3** model, which means you will be performing your image prediction tasks using the "InceptionV3" model generated during your custom training. Find example code below

```
prediction.setModelTypeAsInceptionV3()
```

- **.setModelTypeAsDenseNet()** , This function sets the model type of the image recognition instance you created to the **DenseNet** model, which means you will be performing your image prediction tasks using the "DenseNet" model generated during your custom training. Find example code below

```
prediction.setModelTypeAsDenseNet()
```

- **.setModelPath()** , This function accepts a string which must be the path to the model file generated during your custom training and must corresponds to the model type you set for your image prediction instance. Find example code,and parameters of the function below

```
prediction.setModelPath("resnet_model_ex-020_acc-0.651714.h5")
```

  – *parameter* **model_path** (required) : This is the path to your downloaded model file.

- **.setJsonPath()** , This function accepts a string which must be the path to the JSON file generated during your custom training. Find example code and parameters of the function below

```
prediction.setJsonPath("model_class.json")
```

  – *parameter* **model_path** (required) : This is the path to your downloaded model file.

- **.loadModel()** , This function loads the model from the path you specified in the function call above into your image prediction instance. You will have to set the parameter **num_objects** to the number of classes in your image dataset. Find example code and parameter details below

```
prediction.loadModel(num_objects=4)
```

  – *parameter* **num_objects** (required) : This must be set to the number of classes in your image dataset.

– *parameter* **prediction_speed** (optional) : This parameter allows you to reduce the time it takes to predict in an image by up to 80% which leads to slight reduction in accuracy. This parameter accepts string values. The available values are "normal", "fast", "faster" and "fastest". The default values is "normal"

- **.predictImage()** , This is the function that performs actual prediction of an image. It can be called many times on many images once the model as been loaded into your prediction instance. Find example code,parameters of the function and returned values below

```
predictions, probabilities = prediction.predictImage("image1.jpg", result_count=2)
```

  – *parameter* **image_input** (required) : This refers to the path to your image file, Numpy array of your image or image file stream of your image, depending on the input type you specified.

  —*parameter* **result_count** (optional) : This refers to the number of possible predictions that should be returned. The parameter is set to 5 by default.

– *parameter* **input_type** (optional) : This refers to the type of input you are parse into the **image_input** parameter. It is "file" by default and it accepts "array" and "stream" as well.

—*returns* **prediction_results** (a python list) : The first value returned by the **predictImage** function is a list that contains all the possible prediction results. The results are arranged in descending order of the percentage probability.

– *returns* **prediction_probabilities** (a python list) : The second value returned by the **predictImage** function is a list that contains the corresponding percentage probability of all the possible predictions in the **prediction_results**.

- **.predictMultipleImages()** , This function can be used to perform prediction on 2 or more images at once. Find example code, parameters of the function and returned values below

```
results_array = multiple_prediction.predictMultipleImages(all_images_array,␣
↪result_count_per_image=2)

for each_result in results_array:
    predictions, percentage_probabilities = each_result["predictions"], each_
↪result["percentage_probabilities"]
    for index in range(len(predictions)):
        print(predictions[index] , " : " , percentage_probabilities[index])
    print("-----------------------")
```

– *parameter* **sent_images_array** (required) : This refers to a list that contains the path to your image files, Numpy array of your images or image file stream of your images, depending on the input type you specified.

– *parameter* **result_count_per_image** (optional) : This refers to the number of possible predictions that should be returned for each of the images. The parameter is set to 2 by default.

– *parameter* **input_type** (optional) : This refers to the format in which your images are in the list you parsed into the **sent_images_array** parameter. It is "file" by default and it accepts "array" and "stream" as well.

– *returns* **output_array** (a python list) : The value returned by the **predictMultipleImages** function is a list that contains dictionaries. Each dictionary correspondes the images contained in the array you parsed into the **sent_images_array**. Each dictionary has "prediction_results" property which is a list of athe prediction result for the image in that index as well as the "prediction_probabilities" which is a list of the corresponding percentage probability for each result.

Sample Codes

Find below sample code for custom prediction

```
from imageai.Prediction.Custom import CustomImagePrediction
import os

execution_path = os.getcwd()

prediction = CustomImagePrediction()
prediction.setModelTypeAsResNet()
prediction.setModelPath(os.path.join(execution_path, "resnet_model_ex-020_acc-0.
↪651714.h5"))
prediction.setJsonPath(os.path.join(execution_path, "model_class.json"))
prediction.loadModel(num_objects=4)

predictions, probabilities = prediction.predictImage(os.path.join(execution_path, "4.
↪jpg"), result_count=5)

for eachPrediction, eachProbability in zip(predictions, probabilities):
    print(eachPrediction , " : " , eachProbability)
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search