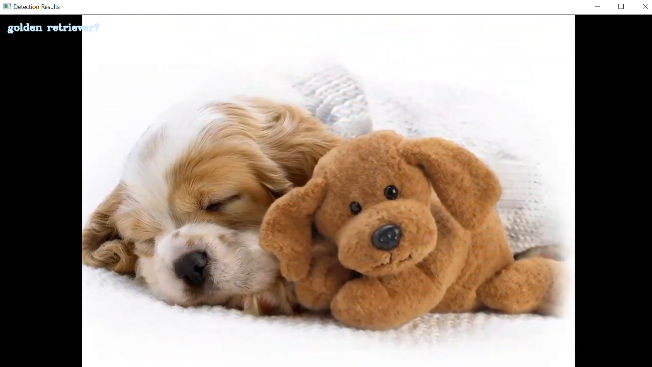
**AI on PC Reference Implementation: How to Build an Object Identification Solution**

**Introduction**

An application capable of identifying the objects of an input video.

**What it does**

This application is one of a series of AI on PC reference implementations aimed at instructing users on how to develop a working solution for a particular problem. It demonstrates how to create an object identification solution using Intel® hardware and software tools. This solution identifies the object present in the input video by classifying it into one of the label class.

**How it Works**

The model uses the Inference Engine included in OpenVINO™. A trained neural network, Squeezenet, classifies objects present in a video by displaying the top ten probabilities along with the confidence for each prediction on the output console.

**Prerequisites**

### [Windows](http://releases.ubuntu.com/16.04/) 10

### OpenVINO™

### Intel Python 3

### Visual Studio 2017, version 15

### [Intel® Movidius™ Myriad™ X VPU](https://www.movidius.com/myriadx)

In this tutorial, you'll learn how to:

* Set up the plug-in dispatcher for the respective device
* Load the network
* Load the network to the plug-in
* Setup the web-camera or video-clip, for the input stream to be identified
* Infer the input
* Post-process the result that ends up in the output buffer and print meaningful results

**Running the application**

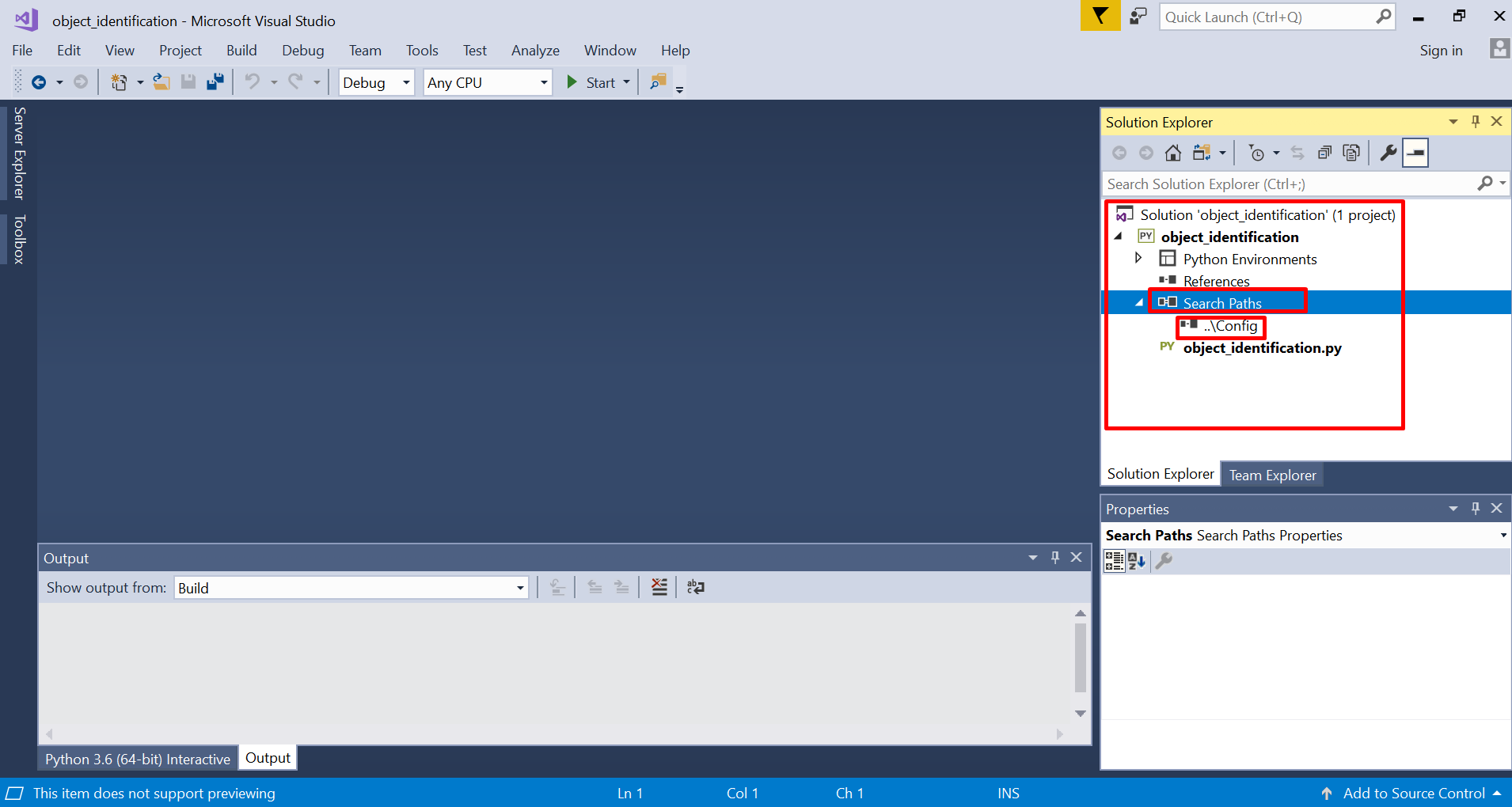
Running this application is as easy as 1-2-3. We start a project in Visual Studio, amend the Python script to infer with the specific inference engine, and run the solution.

**Step 1. Start a project**

To work on the preinstalled demos within Visual Studio, we start by selecting New Project from the File menu, and select the particular code related to our demos. The following is a step-by-step guide to get started quickly.

1. Select **File > New > Project …** to open the **New Project** window.
2. In the left pane, select **Installed > Python**, and in the middle pane, select **From Existing Python code**
3. Modify the project **Name** and **Solution name** to **object\_identification** and click **Browse** button to modify the **Location**
4. Navigate to ‘C:\AI Devkit\Python\_Tutorials\Tutorial3\_ObjectIdentification\ object\_identification, select the object\_identification folder, click **Select Folder** andclick **OK** to complete creating the new python project.
5. In the Create New Project from Existing Python Code click Next to select the object\_identification.py file present in the object\_identification folder. Click **Next > Finish** after selecting the python file to complete the project creation with the existing python code
6. The object\_identification project will load into the Solution Explorer in Visual Studio.
7. For any given project, one has to select the Python environment to be used for execution. In the Solution Explorer, under **<project\_name>**, right-click **Python Environments** and select **Add/Remove Python Environments**. From the displayed list in the pop window, select the environment (in this case, intelpython3) of choice and click **OK**.  The selected environment will appear under the **Python Environments for the project**.
8. There is a config.py reference file that houses the OpenVino logic, it is generic to all the tutorials and is from where the following logical steps take place:
   1. Allows to create the plugin object that will hold the implementation for inference on the Intel® hardware device of our choice (in this case, the VPU plugin) – **2.4**;
   2. Gives access to information about the network model – **2.5**;
   3. Provides the method for the actual inference and identify the object in the frame – **2.10**,
   4. Houses the label information – **2.13**; and
   5. Other functions

We add this reference to our project by right clicking on Search paths of the project -> Add folder to search path. From the pop up window that appears, give the path to the folder containing config.py, Select the Config folder. You have now added the config.py to the project. You can see the Config folder name below the search path of the project. Once we have the codes ready, your folder structure should look like the one below:



**Step 2. Modify the python script to infer input data using the Inference Engine**

We are now ready to have a code walkthrough. Let us get into the details of the **object\_identification.py** file.

1. In **object\_identification.py** the following import statements provide access to the necessary libraries

from \_\_future\_\_ import print\_function

import sys

import cv2

import numpy as np

import config

1. The following lines declare and initialize some variables we’ll use later in the code

cur\_request\_id = 0

next\_request\_id = 1

is\_async\_mode=False

1. We move to parsing command line input arguments by calling the args variable from config.py. In the config.py, you will see that the args variable gets its value from parsing the arguments added to an ArgumentParser() object. (Appendix P1)

args = config.args

1. We now need to create the plugin object that will hold the implementation for inference on the Intel® hardware device of our choice (in this case, the VPU plugin).

We create the plugin by calling the plugin variable from config.py. This variable is an IEPlugin class object initialized and configured by the following arguments -

1. device type (CPU, GPU MYRIAD) and
2. device extensions (if any)

plugin = config.plugin

In the config.py file,

plugin = IEPlugin(device=args.device, plugin\_dirs=args.plugin\_dir)

1. Let us now take a look at those model IR files (the network and the weights) that we have for inference. Call the get\_execnet\_inputstream method from **config.py** to get information about the network model read. The method also loads the network to the plugin, returns the input and the output information, and input dimension (n,c,h,w) from the model directly

exec\_net, input\_blob, out\_blob, input\_stream, n, c, h, w = config.**get\_execnet\_inputstream**(args.model)

The variables have the following purpose –

1. exec\_net – Loads a network that was read from the IR to the plugin and creates an executable network from the network object
2. input\_blob – Holds the input information of the network loaded
3. out\_blob – Holds the output information of the network loaded
4. input\_stream – Holds the current input stream for the given input (video or webcam)
5. n, c, h, w – hold the input frame dimensions (total number, number of channels (RGB), height, width)

In the config.py file,

The get\_execnet\_inputstream returns the input and output information of the network, loads the network to the plugin and returns the input dimensions for the network input

def **get\_execnet\_inputstream**(model\_name):

net = **get\_network**(args.device,model\_name)

plugin = IEPlugin(device=args.device, plugin\_dirs=args.plugin\_dir)

plugin = **get\_plugin**(plugin,net)

assert len(net.inputs.keys()) == 1, "Sample supports only single input topologies"

input\_blob = next(iter(net.inputs))

out\_blob = next(iter(net.outputs))

exec\_net\_name = plugin\_name.load(network=net, num\_requests=2)

n, c, h, w = net.inputs[input\_blob].shape

input\_stream = **get\_input\_stream**()

return exec\_net\_name, input\_blob, out\_blob, input\_stream, n, c, h, w

The get\_input\_stream() method, as the name suggests, sets the input stream to 0 (if it is a camera input) or to the input file provided and returns stream

def **get\_input\_stream**():

if args.input == 'cam':

input\_stream = 0

else:

input\_stream = args.input

assert os.path.isfile(args.input), "Specified input file doesn't exist"

return input\_stream

The get\_network method returns an IENetwork object that contains the information about the network model read from IR. If the model arguments are empty, an empty string is returned instead.

def **get\_network**(device\_name,model\_name):

if device\_name:

model\_name\_xml = model\_name

model\_name\_bin = os.path.splitext(model\_name\_xml)[0] + ".bin"

netEnabled = IENetwork(model=model\_name\_xml, weights=model\_name\_bin)

else:

netEnabled = ""

return netEnabled

The get\_plugin method adds extension file, if available, for the specified device, checks for unsupported layers in the network (if any) and returns the modified plugin object.

def **get\_plugin**(plugin,net):

if args.cpu\_extension and 'CPU' in args.device:

plugin.add\_cpu\_extension(args.cpu\_extension)

#If you are running in CPU, Please uncomment the below lines.

if plugin.device == "CPU":

supported\_layers = plugin.get\_supported\_layers(net)

not\_supported\_layers = [l for l in net.layers.keys() if l not in supported\_layers]

if len(not\_supported\_layers) != 0:

log.error("Following layers are not supported by the plugin for specified device {}:\n {}".

format(plugin.device, ', '.join(not\_supported\_layers)))

log.error("Please try to specify cpu extensions library path in demo's command line parameters using -l "

"or --cpu\_extension command line argument")

sys.exit(1)

return plugin

1. We will now handle the video streaming for the input. The input could be a webcam setup on the laptop or an offline video from the local storage.

cap = cv2.VideoCapture(input\_stream)

1. The following code logs the current state of the program to the standard stream.

log.info("Starting inference in async mode...")

log.info("To switch between sync and async modes press Tab button")

log.info("To stop the demo execution press Esc button")

1. We are now ready to begin processing each frame (within a while loop) from the input stream

while cap.isOpened():

1. At the top of the loop we grab the frame and check if the frame has been fetched or not. If it has not fetched the frame, the loop would terminate.

ret, frame = cap.read()

if not ret:

break

1. We can now start the actual inference and identify the object in the frame. The next call is to the exe\_start\_async method from the **config.py** file. The method returns the InferRequest handler that starts the inference for the specified request

if is\_async\_mode:

exec\_net = config.**exec\_start\_async**(exec\_net, n, c, h, w, next\_frame, next\_request\_id, input\_blob)

else:

exec\_net = config.**exec\_start\_async**(exec\_net, n, c, h, w, frame, cur\_request\_id, input\_blob)

From the config.py file,

def **exec\_start\_async**(exec\_net\_recognition, n, c, h, w, frame\_passing, id\_of\_request, input\_blob\_name):

in\_frame = cv2.resize(frame\_passing, (w, h))

in\_frame = in\_frame.transpose((2, 0, 1)) # Change data layout from HWC to CHW

in\_frame = in\_frame.reshape((n, c, h, w))

exec\_net\_recognition.start\_async(request\_id=id\_of\_request, inputs={input\_blob\_name: in\_frame})

return exec\_net\_recognition

1. We now process the results for displaying the output given that the inference result for the frame has been obtained. The *if* condition is to verify if the InferRequest results have been obtained for further processing

if exec\_net.requests[cur\_request\_id].wait(-1) == 0:

1. It is go time for the post processing of the results. As a first step here, we store the result value, after which we get the OutputInfo result from the outputs dictionary into a local variable.

res = exec\_net.infer(inputs={input\_blob: in\_frame})

1. We now check if we have the result value from the previous step and get the top most index from the result .We also find the corresponding label, which indicates the identified object. After this, our task is to filter out any prediction that is below a pre decided confidence level. So the given logic sets the label to the frame if the confidence value of the inference is greater than a probability threshold value.

if res != "":

top\_ind = np.argsort(res[out\_blob], axis=1)[0, -args.number\_top:][::-1]

labels = config.**get\_labels\_object\_detection**()

# comparing with the threshold

if res[out\_blob][0, top\_ind[0][0][0]][0][0] > args.threshold:

cv2.putText(frame, str(labels[top\_ind[0][0][0]]), (15, 30),

cv2.FONT\_HERSHEY\_COMPLEX, 0.6, (255, 235, 205), 2)

In the config.py file,

def **get\_labels\_object\_detection**():

label\_file\_name = os.path.splitext(args.model)[0] + ".labels"  
 labels = []  
 # append the labels to a list named labels  
 with open(label\_file\_name, "r") as inputFile:  
 for strLine in inputFile:  
 strLine.strip()  
 strLine.splitlines()  
 labels.append(strLine)  
 return labels

The function returns a category label map for the detected label.

1. Now that all the processing is done, we can view the detected object label on the input image frame by displaying each frame.

cv2.imshow("Detection Results", frame)

1. We have the provision for the user to switch the processing mode from asynchronous to synchronous and vice versa. This should happen when the user hits the tab button (ASCII key 9) during the execution. An escape keypress (ASCII key 27) on the other hand should lead to exiting the execution. The following code captures the two actions

key = cv2.waitKey(1)

if key == 27:

break

if 9 == key:

is\_async\_mode = not is\_async\_mode

log.info("Switched to {} mode".format("async" if is\_async\_mode else "sync"))

1. As the last step in the logic, we finish the script by closing open windows and releasing all the pointers thus freeing up the memory space

cv2.destroyAllWindows()

del exec\_net

del plugin

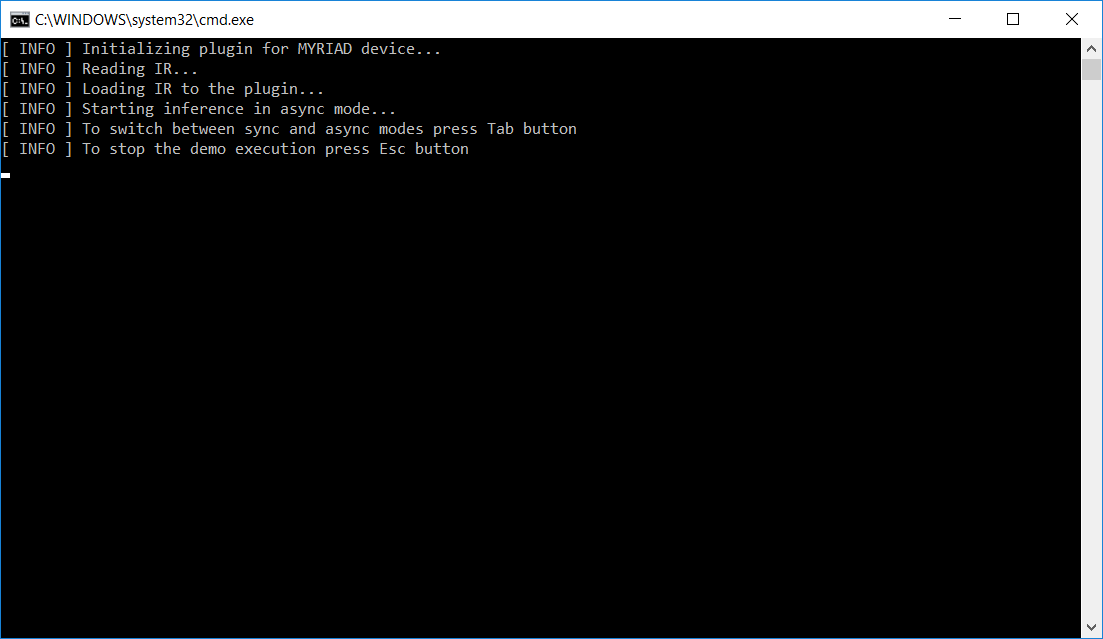
Congratulations you have successfully completed the section for understanding the inference code!

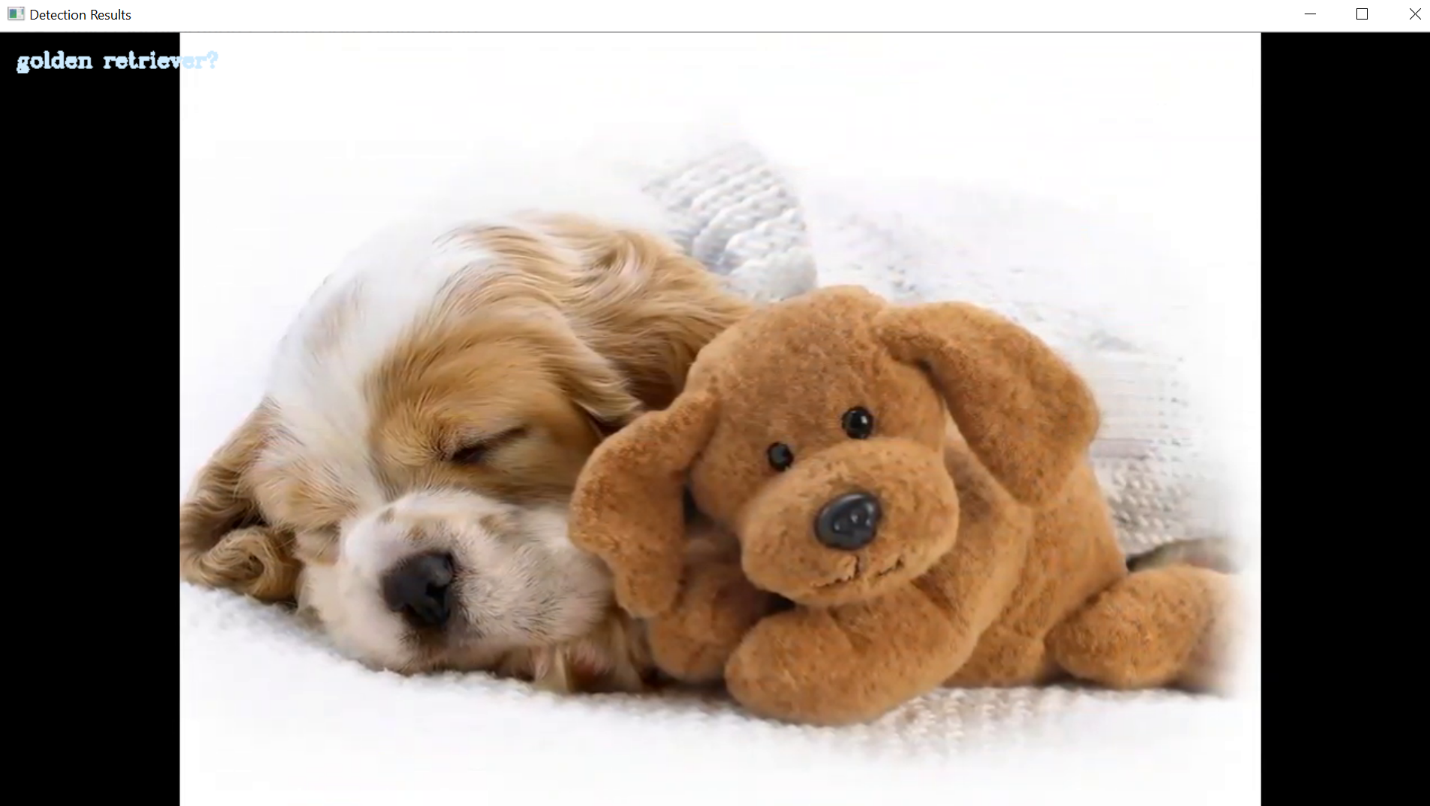
It’s time to see our object detection code in action!

**Step 4. Run the solution**

1. Right click the **object\_identification** project from the Solution Explorer and click Properties
2. The **General** properties for the project opens
3. Click the **Debug** tab
4. Click the text box to the right of **Script Arguments** and enter the following arguments

*-m C:\AI Devkit\Python\_Tutorials\Tutorial3\_ObjectIdentification\model\_files\FP16\squeezenet1.1.xml -i C:\AI Devkit\Python\_Tutorials\Tutorial3\_ObjectIdentification\model\_files\objects.mp4 -d MYRIAD -thresh 0.4*

1. Press Ctrl + F5 to run the program. A terminal should open displaying the progress and the results would be displayed in a streaming video frame 

****

1. Press “Esc” to exit

**APPENDIX**

1. STEP 2.5 - calling the args variable from **config.py**

args = config.args

In the config.py file,

args = build\_argparser().parse\_args()

def build\_argparser():

parser = ArgumentParser()

# parser.add\_argument("-h", "--help", help="Print a usage message.")

parser.add\_argument("-m", "--model", help="Path to an .xml file with a trained model.", required=True, type=str)

parser.add\_argument("-m\_em", "--model\_em", help="Path to an .xml file with a trained model.",

type=str, default=None)

parser.add\_argument("-m\_va", "--model\_va", help="Path to an .xml file with a trained model.", required=False, type=str)

parser.add\_argument("-m\_lpr", "--model\_lpr", help="Path to an .xml file with a trained model.", required=False, type=str)

parser.add\_argument("-i", "--input",

help="Path to video file or image. 'cam' for capturing video stream from camera", required=True,

type=str)

parser.add\_argument("-n\_ag", "--n\_age",

help="Path to an .xml file with a trained Age/Gender Recognition model. "

"impl.", type=str, default=None)

parser.add\_argument("-n\_hp", "--n\_hp", help="Path to an .xml file with a trained Head Pose Estimation model.",

type=str, default=None)

parser.add\_argument("-l", "--cpu\_extension",

help="MKLDNN (CPU)-targeted custom layers.Absolute path to a shared library with the kernels "

"impl.", type=str, default=None)

parser.add\_argument("-c", "--clDNN", help="Absolute path to clDNN (GPU) custom layers config (\*.xml).")

parser.add\_argument("-pp", "--plugin\_dir", help="Path to a plugin folder", type=str, default=None)

parser.add\_argument("-d", "--device",

help="Specify the target device to infer on; CPU, GPU, FPGA or MYRIAD is acceptable. Demo "

"will look for a suitable plugin for device specified (CPU by default)", default="CPU",

type=str)

parser.add\_argument("-d\_va", "--device\_va",

help="Specify the target device to infer on; CPU, GPU, FPGA or MYRIAD is acceptable. Demo "

"will look for a suitable plugin for device specified (CPU by default)", type=str)

parser.add\_argument("-d\_lpr", "--device\_lpr",

help="Specify the target device to infer on; CPU, GPU, FPGA or MYRIAD is acceptable. Demo "

"will look for a suitable plugin for device specified (CPU by default)", type=str)

parser.add\_argument("--labels", help="Labels mapping file", default=[], type=list)

parser.add\_argument("-pt", "--prob\_threshold", help="Probability threshold for detections filtering",

default=0.5, type=float)

parser.add\_argument("-thresh", "--threshold", help="Confidence threshold for classification",

default=0.3, type=float)

parser.add\_argument("-nt", "--number\_top", help="Number of top results", default=10, type=int)

parser.add\_argument("-ni", "--num\_iter", help="Number of iterations", default=1)

parser.add\_argument("-pc", "--perf\_report", help="Enables per-layer performance report")

parser.add\_argument("-p\_msg", "--msg\_plugin", help="Enables messages from a plugin")

parser.add\_argument("-nireq", "--num\_iter\_req", help="Number of infer request for pipelined mode", default=1)

parser.add\_argument("-dyn", "--dyn\_batch\_size", help="Enable dynamic batch size")

parser.add\_argument("-iout", "--iou\_threshold", help="Intersection over union threshold for overlapping detections"

"filtering", default=0.4, type=float)

parser.add\_argument("-async", "--async", help="Enable asynchronous mode")

parser.add\_argument("-no\_wait", "--no\_wait", help="Do not wait for key press in the end")

parser.add\_argument("-no\_show", "--no\_show", help="Do not show processed video")

parser.add\_argument("-r", "--r", help="Output inference results as raw values")

parser.add\_argument("-t", "--t", help="Probability threshold for detections")

return parser