A

Interim Project Report

On

“Car Detection”



(Academic Session 2020-21)

Mentor: Submitted by:

Ms. Vibha Santhanam Lovesh Mittal

Greeshma K G

Group:10 Medha MH

Ayesha Fatima Deepa Grace

Abstract

With the advancement in image recognition algorithms, applications of image recognition have become increasingly common in our lives (Captcha, government, surveillance, etc.). One such application is for image classification tasks, which common uses included digit recognition and organism classification. In this paper, the authors present an empirical comparison of different machine learning techniques in order to build a vehicle model classification model, which could be used in various business applications such as a mobile app to identify car models based on pictures taken by users.

Car detection and identification is an important task in the area of traffic control and management. Typically, to tackle this task, large datasets and domain-specific features are used to best fit the data. In our project, we implement, train, and test several state-of-the-art classifiers trained on domain general datasets for the task of identifying the make and models of cars from various angles and different settings, with the added constraint of limited data and time. We experiment with different levels of transfer learning for fitting these models over to our domain. We also perform object detection like masked RCNN and make a interactable GUI interface. We report and compare these results of various models, and discuss the advantages of this approach.

Introduction

With technology development and image recognition methods being increasingly accurate, many business and government applications arise. Among them, recognition technologies are often used for security and/or tracking purposes. The Stanford Car Dataset will be utilized to build a vehicle recognition predictive model. The ultimate goal of the model is to classify a car’s make and model given an input image. This model could be further developed to be used in creating a mobile application that assists users in identifying cars of interest. The users would simply take a picture of the vehicle of interest and the application would return information (Make and Model) regarding the recognized vehicle. Users could also input a picture found on the Internet or elsewhere. This application is in fact currently being used by vehicle renting/sharing companies such as Bird and Lime. Partnerships with other car dealership websites could be beneficial in enhancing the application quality, since the recognized vehicle name would be used in searching the partners’ database to obtain valuable information such as availability, price and so on. An improved model would result in direct reviews/subscription profit. This application could help people who are not familiar with cars or who simply want quick information without searching the Internet themselves. Another potential development idea of this project would be for traffic law enforcement. Traffic AI is a huge market globally. One example of this would be the China Transinfo Technology Corp. They focus on extracting features of vehicles the moment they appear in security cameras, which can help the police to track the targeted cars.

*About the dataset*:

The Stanford Cars Dataset is a large collection of vehicle images produced by Dr. Jonathan Krause and his team at Stanford University. To generate an initial list of car labels, the authors crawled an unspecified popular car website to create a list of all cars from 1990 to 2012. Because many car models do not change their appearances across model years, the authors merged car classes with similar visual features using a technique called perceptual hashing. This technique compares two media objects such as images to see if they are different from each other. In this study, the authors used Hamming distance, the difference between two strings of numbers that represent the pictures, as a measure to determine the dissimilarities between car classes. After the initial round of perceptual hashing, the authors generated 197 classes of cars. To expand on the pool of car images, Dr. Krause and his team collected car images from Flickr, Google, and Bing. While these search engines allowed the authors to collect many images, they needed to verify that the collected car images were from the correct car classes. To verify the identities of these images, the authors utilized Amazon Mechanical Turk (AMT) workers to annotate the car images with the correct car labels. The car identification task contained an image of the car that needed to have its identity verified, an image of the actual car from the class of interest, and an image of a car from a different class that could easily be mixed up as an image from the target class. Based on the two images with confirmed classes, the workers must decide whether the unverified car image was from the class of interest. If not, the workers annotated the image with the correct class label. For the workers to qualify for this task, they needed to pass a series of tests that contained some of the most difficult cars to identify.

To determine the quality of annotations, the authors used a technique called Get Another Label (GAL), which is an algorithm using expectation-maximum, a type of maximum-likelihood algorithm that estimates values for model parameters for incomplete data, that estimates the probability that an image is from a certain class while also determining the quality of a worker based on their correct annotations. The criteria for GAL for the car annotation task were: 1) an agreement of workers on the correct car class of an image and 2) the ability of workers to identify “gold standard” images, which were images that the authors knew the correct labels. After the GAL probabilities of a candidate image exceeded a certain threshold, the image was put into the target class. GAL was also used to further weed out poor quality workers by assigning more and more images to users that have low scores, discouraging them to continue the task. After obtaining the set of images with assigned classes, the authors utilized a different group of AMT workers to assign bounding boxes, the section of an image that contains the target object, using a technique presented by Fei-Fei et. al. To further remove duplicate images, the authors used another round of perceptual hashing on the images based on the bounding boxes, yielding a total of 16,185 images with 196 classes of cars.

Problem Statement

*PROJECT OBJECTIVE*: Design a DL based car identification model.

*DOMAIN*: Automotive. Surveillance.

*CONTEXT*: Computer vision can be used to automate supervision and generate action appropriate action trigger if the event is predicted from the image of interest. For example, a car moving on the road can be easily identified by a camera as make of the car, type, colour, number plates etc.

*DATA DESCRIPTION*: The Cars dataset contains 16,185 images of 196 classes of cars. The data is split into 8,144 training images and 8,041 testing images, where each class has been split roughly in a 50-50 split. Classes are typically at the level of Make, Model, Year, e.g. 2012 Tesla Model S or 2012 BMW M3 coupe.

‣Train Images: Consists of real images of cars as per the make and year of the car.

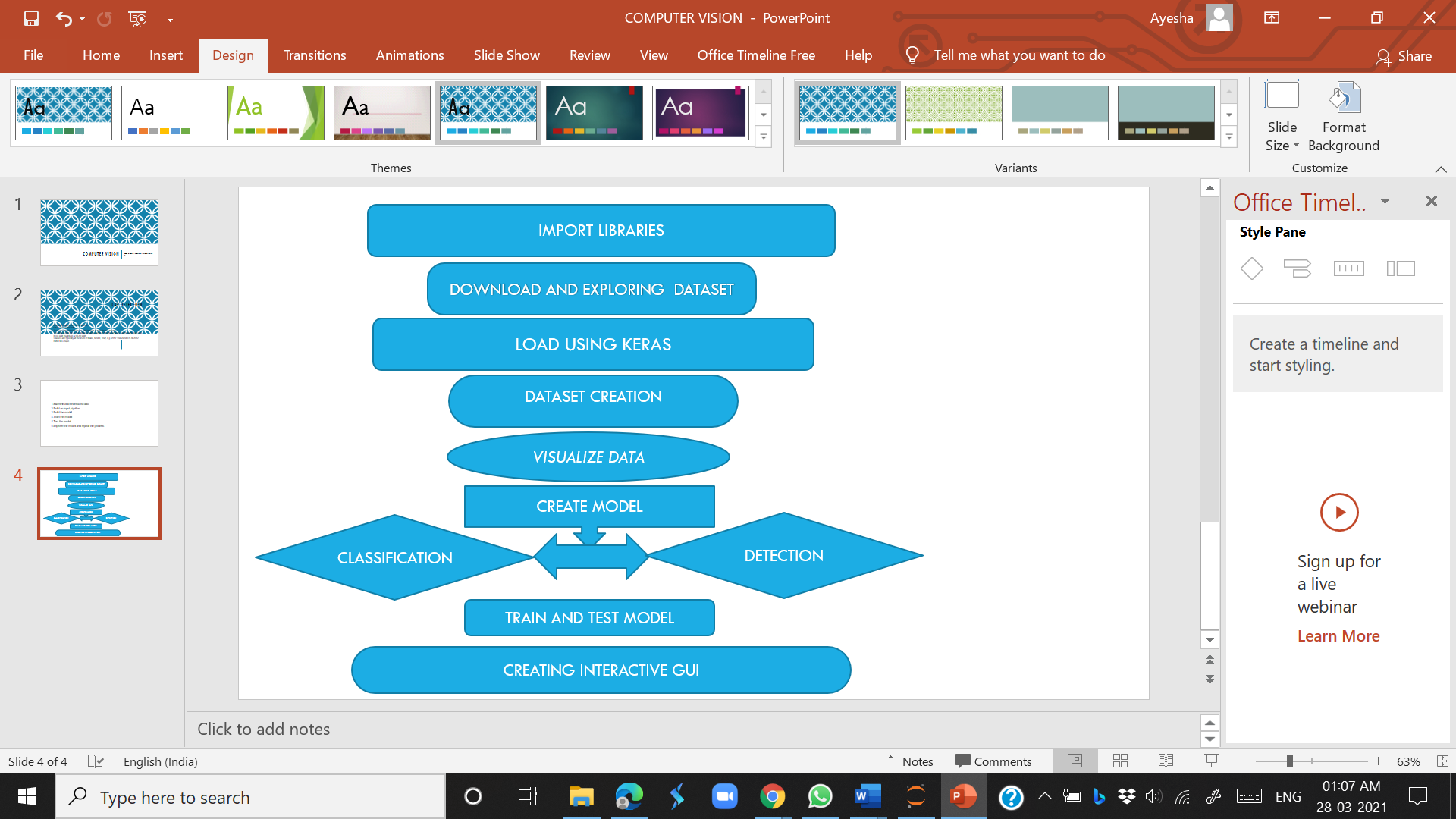
‣Test Images: Consists of real images of cars as per the make and year of the car.

‣Train Annotation: Consists of bounding box region for training images.

‣Test Annotation: Consists of bounding box region for testing images

WORK FLOW PATH

1. Examine and understand data
2. Build an input pipeline
3. Build the model
4. Train the model
5. Test the model
6. Improve the model and repeat the process



Data Preprocessing & Exploratory Data Analysis

The Cars dataset contains 16,185 images of 196 classes of cars. The data is split into 8,144 training images and 8,041 testing images, where each class has been split roughly in a 50-50 split. Classes are typically at the level of Make, Model, Year, e.g. 2012 Tesla Model S or 2012 BMW M3 coupe.

We have three files:

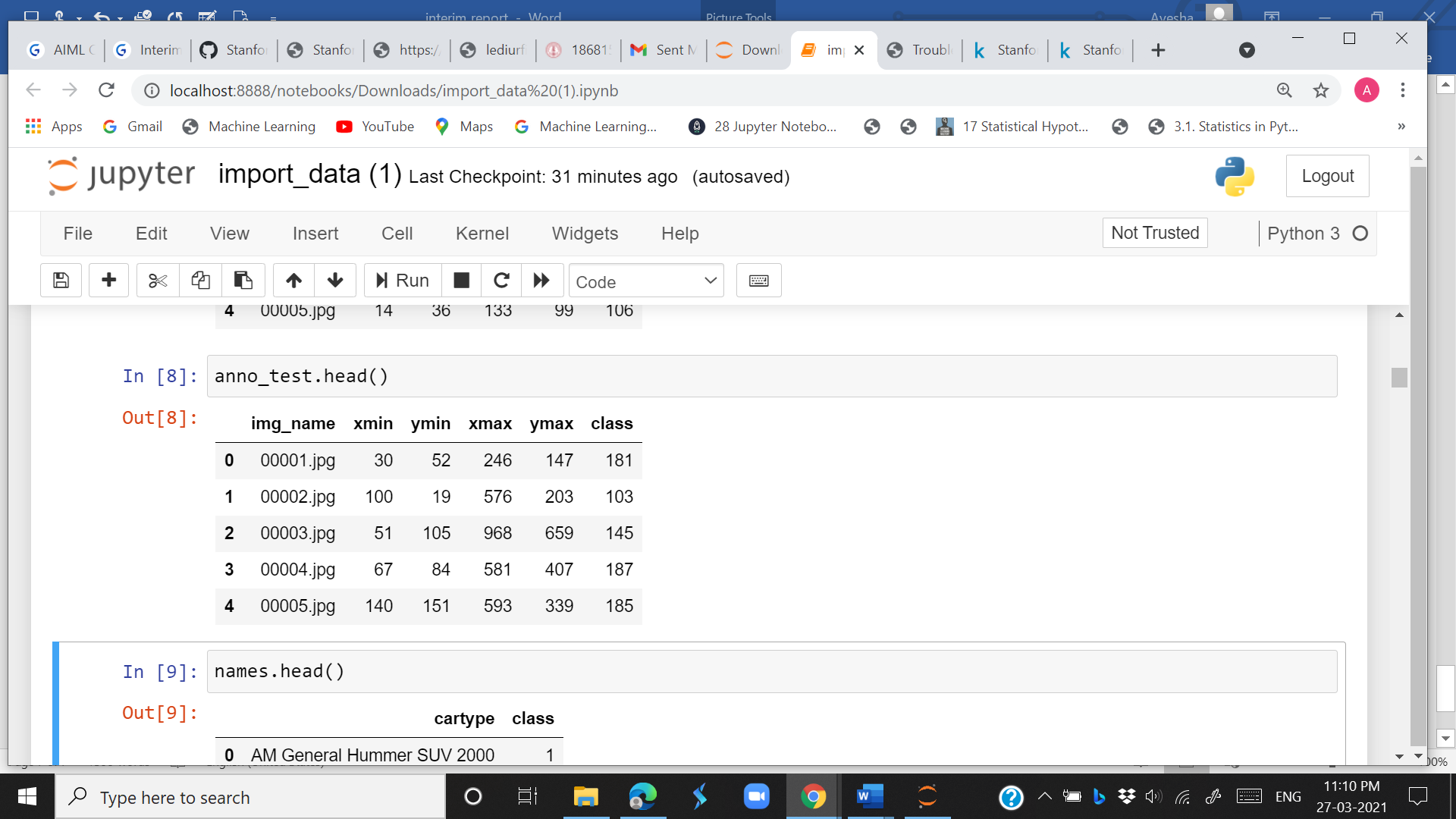
1)anno\_train

2)anno\_test

Both of these contain a column for image name (00001.jpg), and five other numerical columns. We assign names for each column as image\_name, xmin, ymin, xmax, ymax and class

anno\_train = pd.read\_csv('CARS/anno\_train.csv',names=['img\_name','xmin','ymin','xmax','ymax','class'])

anno\_test = pd.read\_csv('CARS/anno\_test.csv',names=['img\_name','xmin','ymin','xmax','ymax','class'])

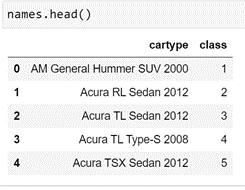
 

3)class

This file contains a column having names of cars. We then assign names for columns as car\_type and the index represents the class number. Since the index starts from 0 in python but our class number starts from 1, we increase index by 1

names = pd.read\_csv('CARS/names.csv',names=['cartype'])

names.loc[:,'class'] = names.index+1



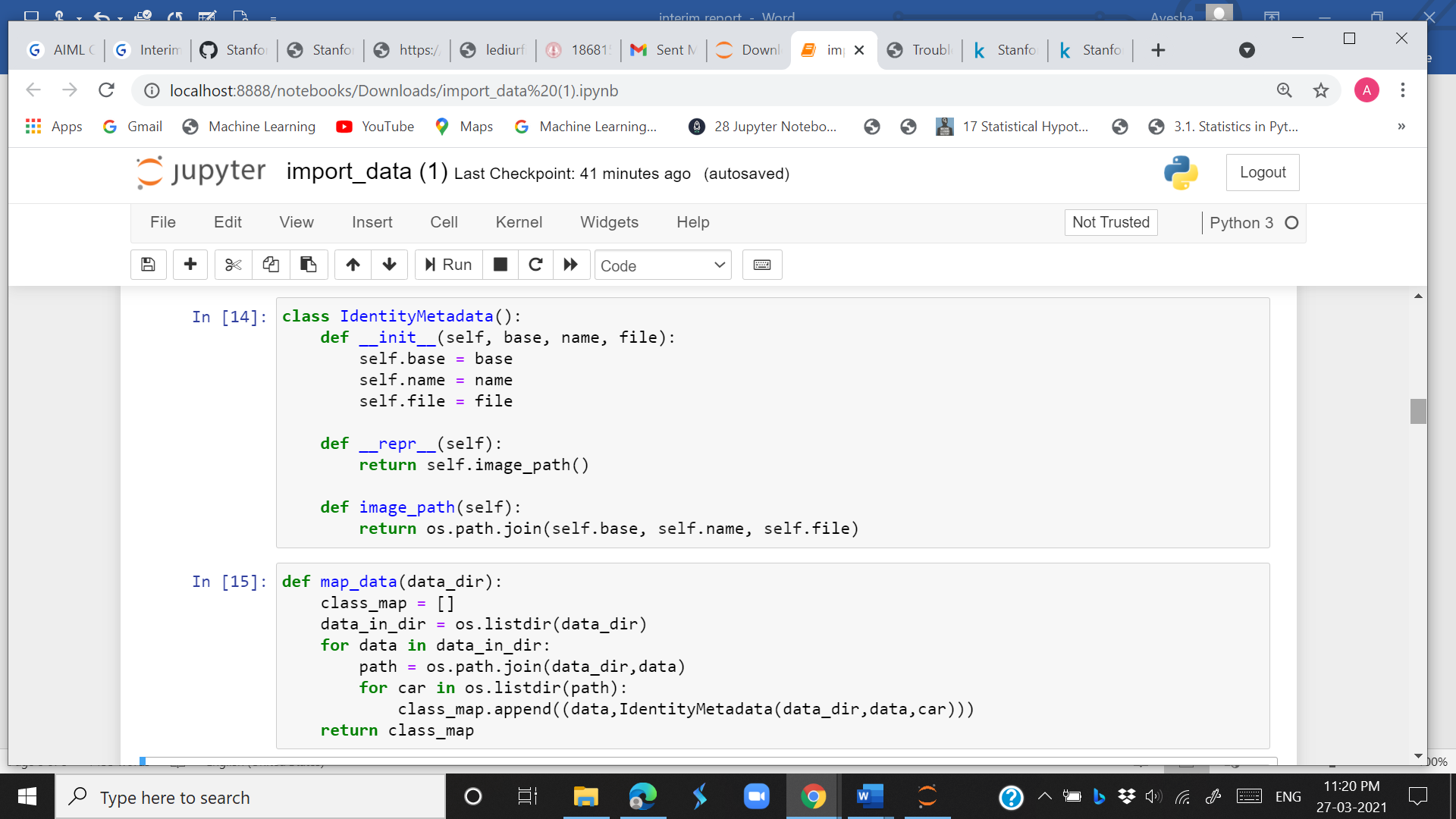
There was a discrepancy in the car name for Ram C/V Cargo Van Minivan 2012 which was written as Ram C-V Cargo Van Minivan 2012. Both the names were equated as same

names.loc[names.loc[:,'cartype'] == 'Ram C/V Cargo Van Minivan 2012','cartype'] = 'Ram C-V Cargo Van Minivan 2012'

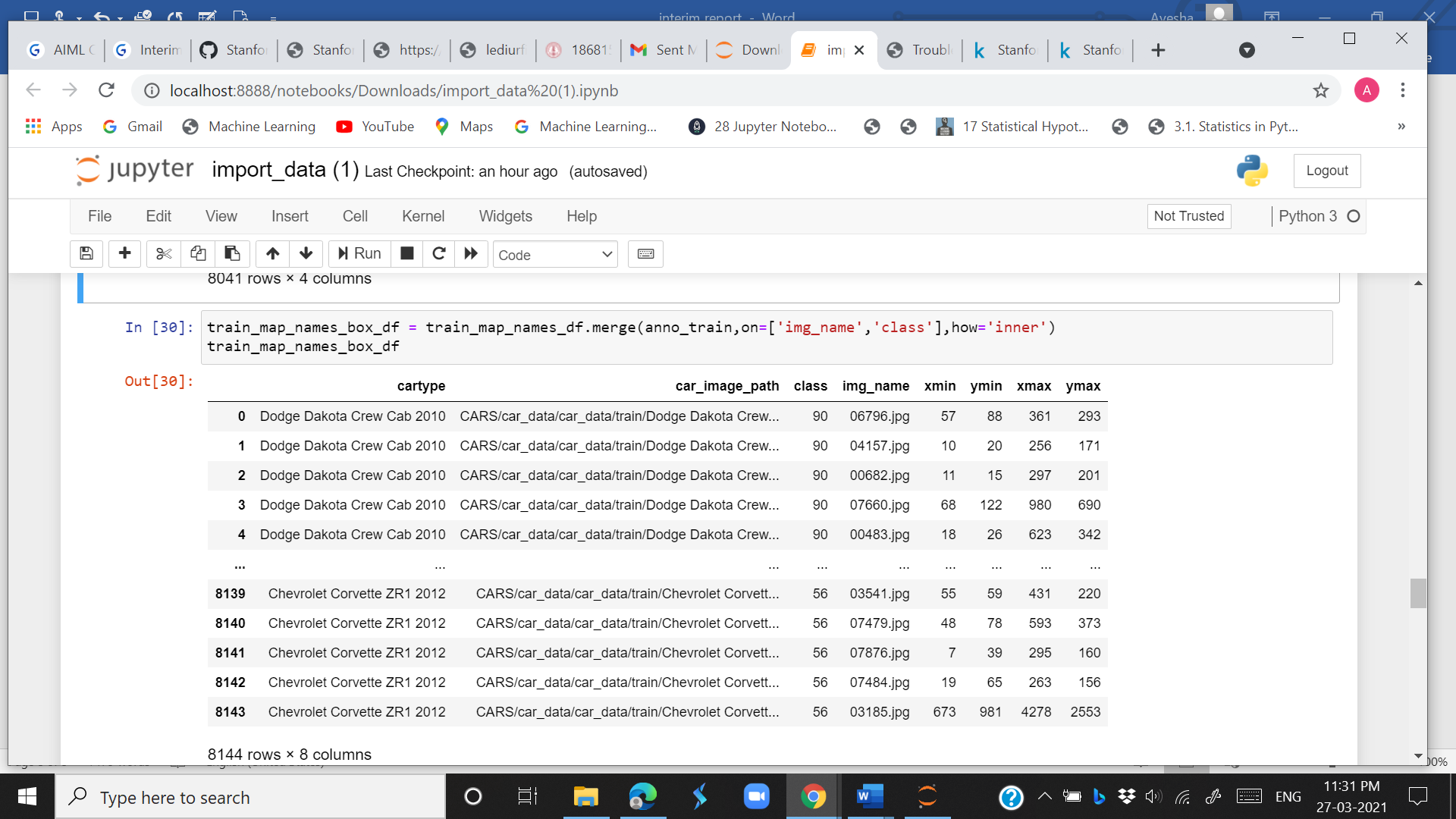
The dataset has no null or missing values.

The shape of image was checked by calling images using its path.

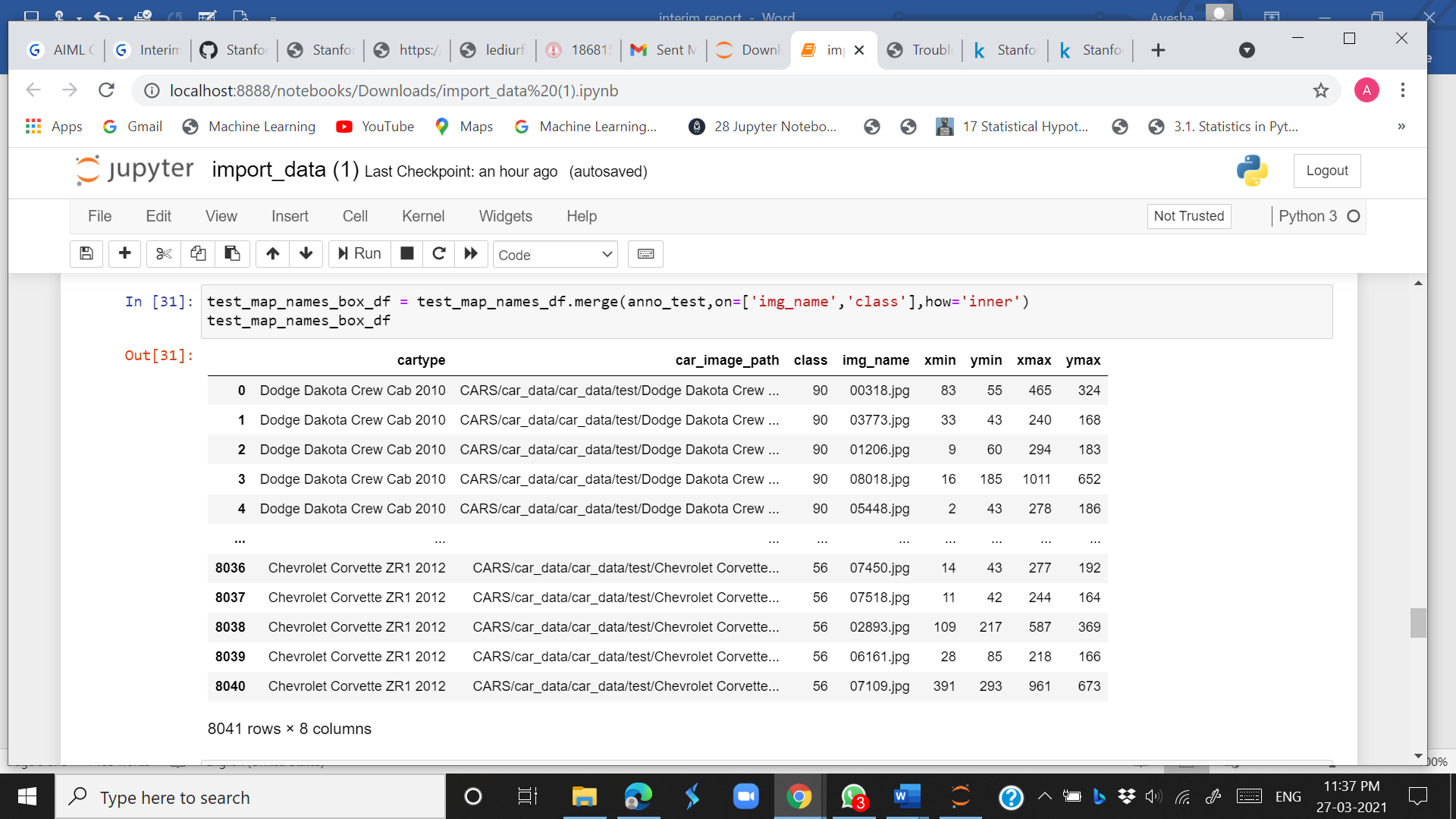
The image paths were then mapped to the created train and test directories.



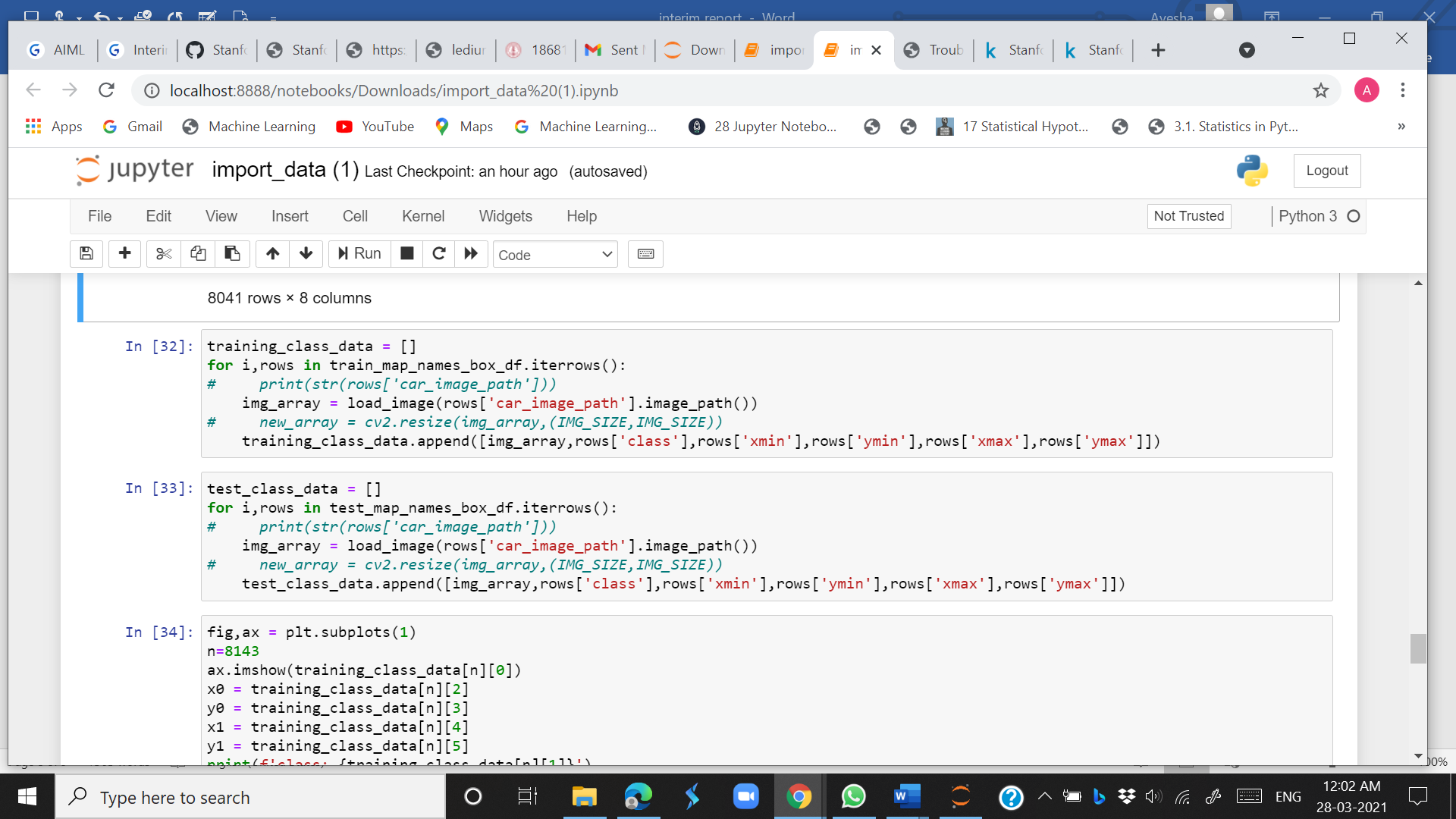
The mapped training data was merged with class of each image and the anno\_train dataset to create the train\_map\_names\_box\_df



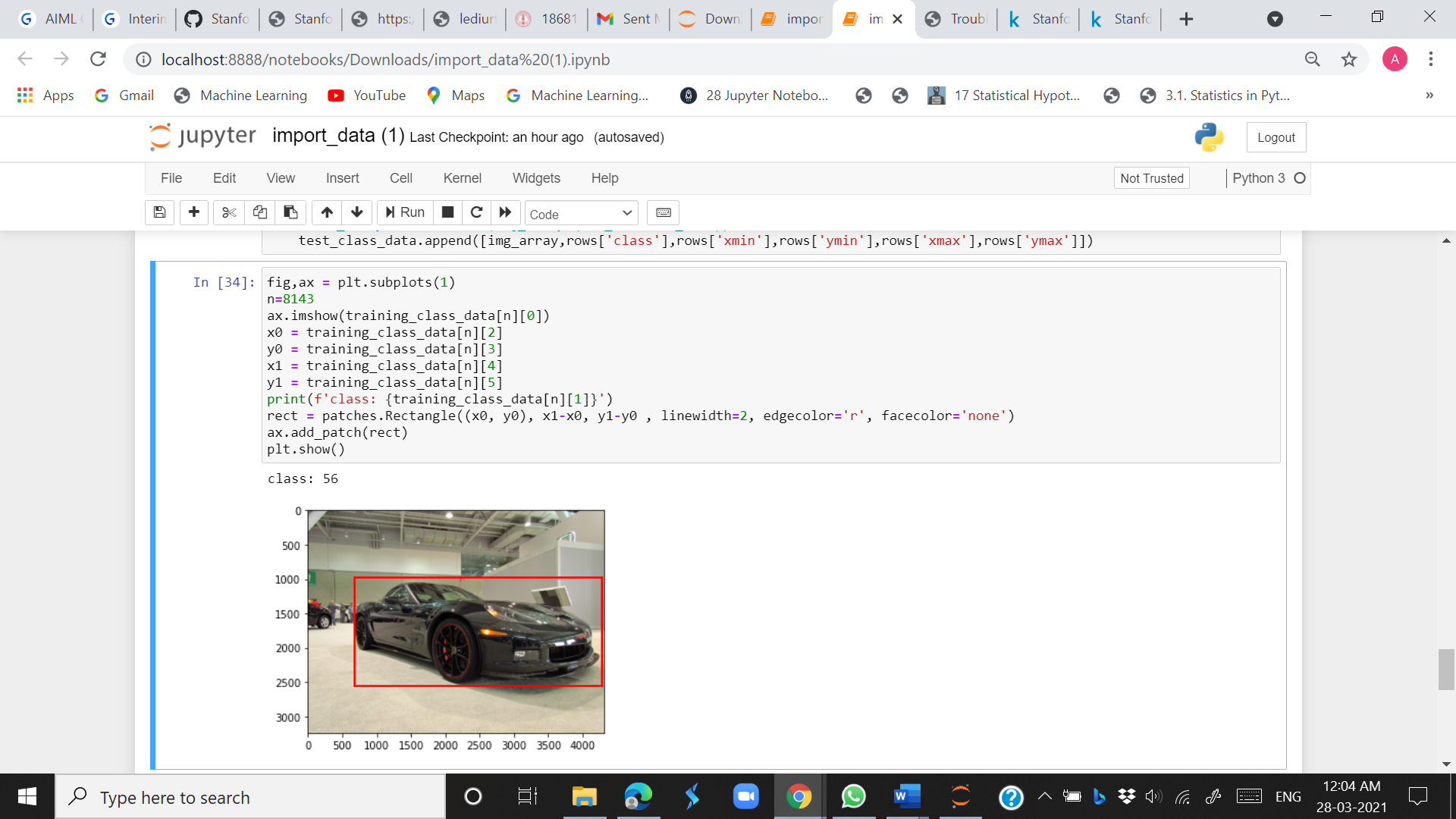
The mapped test data was merged with class and image\_name of each image and anno\_test dataset to create the test\_map\_names\_box\_df.



The train and test data was next loaded but due to large number of classes and images the time taken and disk requirement was very large so the data was divided and then loaded to get training\_class\_data and test\_class\_data



The image was then loaded with bounding boxes



The X and Y for train and test data was created.

X\_train = training\_class\_data[0]

X\_test = test\_class\_data[0]

y\_train = training\_class\_data[1:]

y\_test = test\_class\_data[1:]

Preprocessing and Model Building

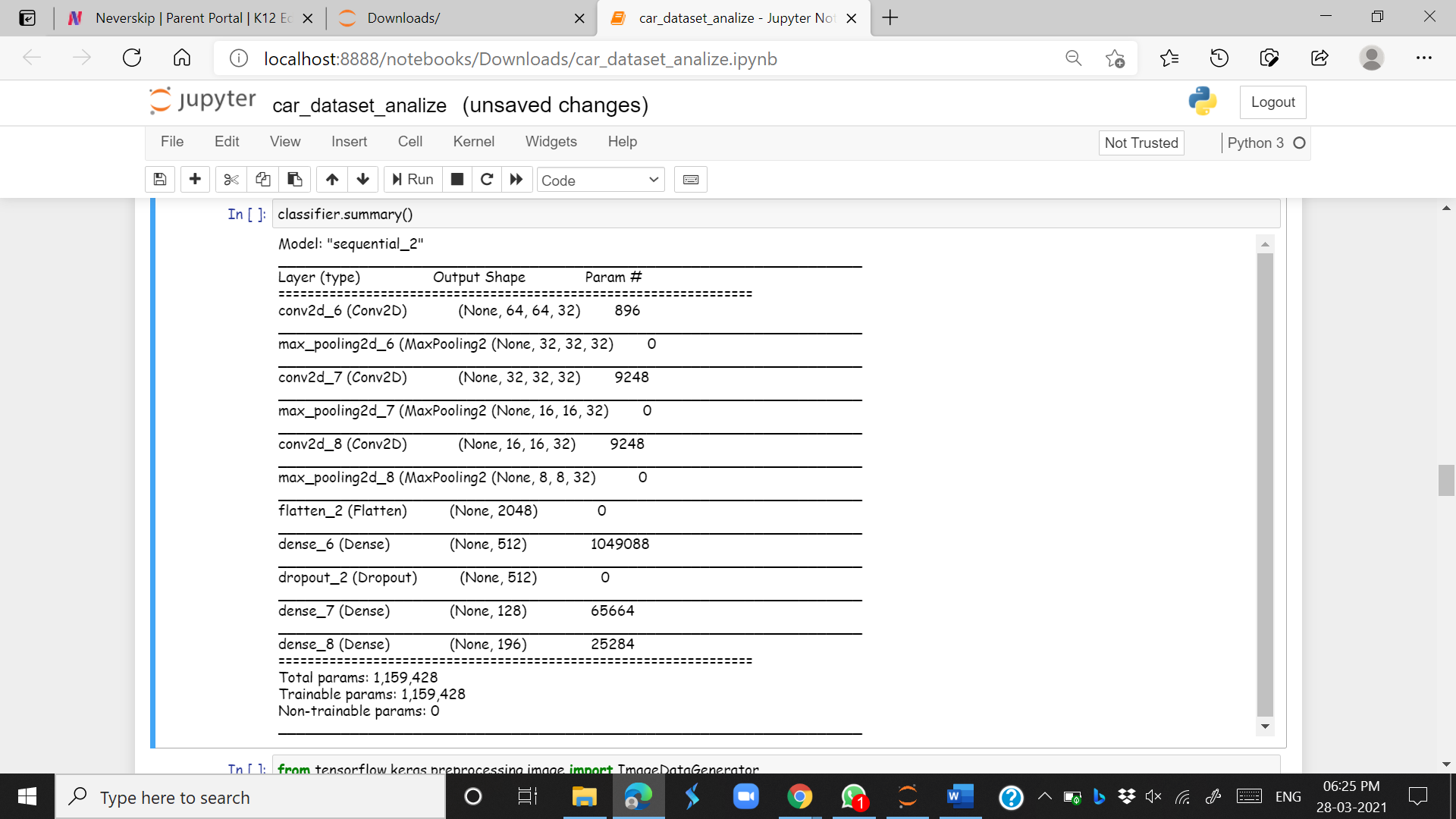
The two approaches i.e., Classification models and Detection models are being explored. We are using classification algorithms, CNNs and ResNet and training our data on it and comparing out train and test accuracy. Further we are going for MASKED CNN and MOBILENET object detection algorithms to see if we can further improve accuracy.

*Classification Models*

CNN

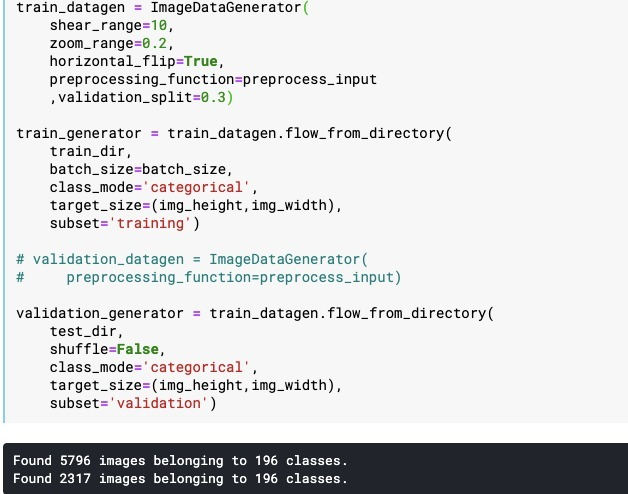
Convolutional neural networks (CNNs) are an extension of neural nets adapted for image classification. The model is composed of layers of filters and compressions in sequence. Filter weights are adjusted like those in the neural net for similar purposes. With an appropriate amount of training samples and adequately-tuned parameters, a CNN will automate the feature selection process and accurately classify the input. Convolution layers contain trainable kernels (sets of weights) which transform patches of the input image. These are essentially different kinds of image filters that the model adjusts during training in order to optimize the chosen loss function. A well-trained model contains an array of filters that effectively discriminate between different classes of images. In the Keras package, convolutional layers have several user-defined inputs such as stride length and kernel dimensions. Altering these two parameters in particular can influence model learning and performance, but were not explored in depth here. The default kernel size of 3x3 was used for each models’ filter layers. Max-pooling is used after each block of convolution layers. The pooling process summarizes a window of pixels into a single output value. In the case of max-pooling, the window will be condensed into the highest value. This helps emphasize and maintain areas of large contrast from layer to layer.

The custom convolutional neural networks is as below:



This custom CNN was trained after tuning the learning rate and fit on regular Stanford Cars images (under-sampled). We used batch size of 32 and used 20 epochs. The model had the highest validation accuracy of about 10% post learning rate tuning with a test accuracy of 6%. When the number of epochs was further increased, the training accuracy increases to 90%+ while validation accuracy is 10%, leading to overfitting. This illustrates that simple CNN models are not able to perform well on this complex and non-structured car image data. This is due to the large number of classes and less number of images per class.

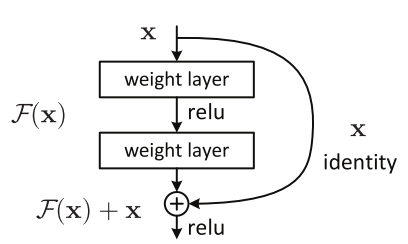
To combat this problem, we have gone for image generation and *Image Augmentation* preprocessing method. The further models are trained with synthetic images. These images are slightly distorted (mirrored, rotated, and sheared to some extent) in order to help the model better generalize the input space. Consequently, this helps combat overfitting since the model is exposed to more than just the ‘real’ training images. Test images are left unmodified in order to evaluate models with actual image data.



We go for *Transfer Learning* of the CNN for better results. It is a machine learning technique that focuses on repurposing learned classifiers for new tasks. In transfer learning for CNNs, a base network is trained on a base dataset to create weights and features. This classifier is then transferred to a new dataset by retraining a subset of the base network’s learned weights and features. The overall effect is a classifier that fits the new dataset with significantly less work than retraining a new network. When the target dataset is significantly smaller than the base dataset, transfer learning can be a powerful tool to enable training a large target network while minimizing overfitting.

ResNet

ResNet, which was proposed in 2015 by researchers at Microsoft Research. The core idea of ResNet is introducing a so-called “identity shortcut connection” that skips one or more layers, as shown in the following figure, called as a Residual Block of the ResNet architecture:

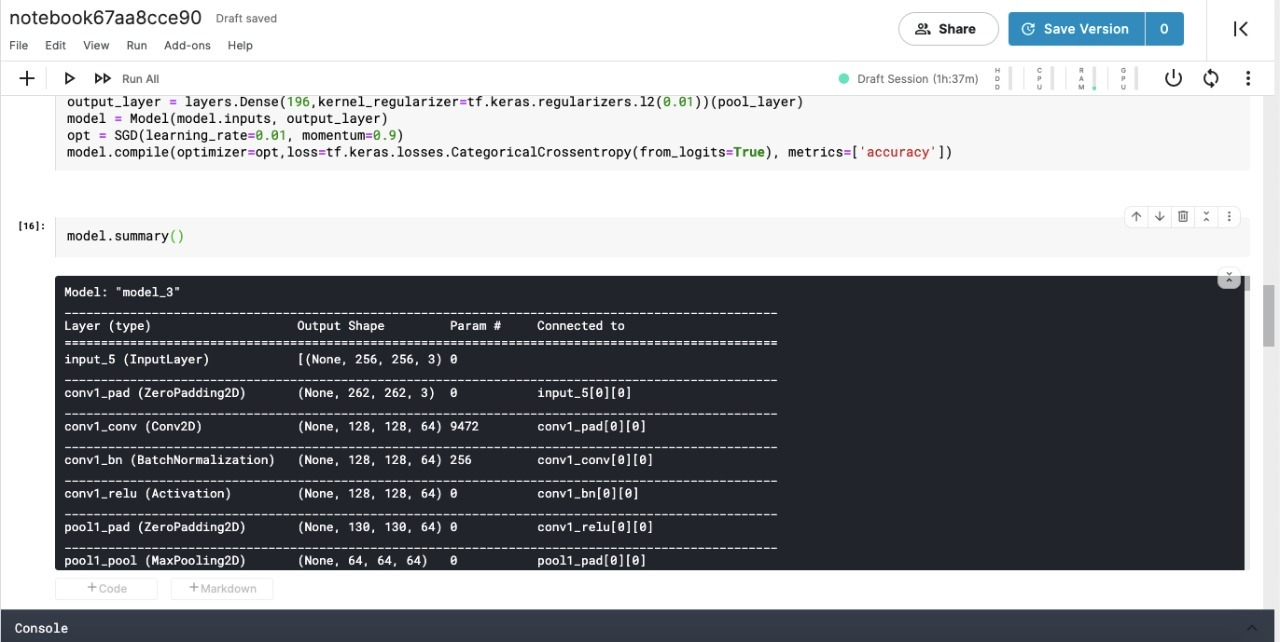


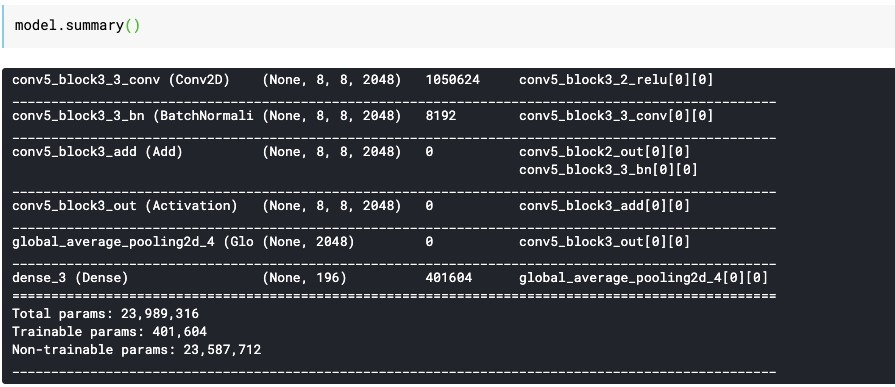
In order to solve the problem of the vanishing/exploding gradient ResNets were introduced. The advantage of adding this type of skip connection is because if any layer hurt the performance of architecture then it will be skipped by regularization. So, this results in training very deep neural network without the problems caused by vanishing/exploding gradient.

Using the Tensorflow and Keras API, we can design ResNet architecture (including Residual Blocks) from scratch.

Below is the implementation of the ResNet architecture we’ve used:







When training using 20 epochs, we get an accuracy of 45% with val\_accuracy of 28% which is much better than CNN results without overfitting.

*Object Detection*

Faster RCNN

**R-CNN** is the first step for Faster R-CNN. It uses **search selective** to find out the regions of interests and passes them to a ConvNet. It tries to find out the areas that might be an object by combining similar pixels and textures into several rectangular boxes. The R-CNN paper uses 2,000 proposed areas (rectangular boxes) from search selective. Then, these 2,000 areas are passed to a pre-trained CNN model. Finally, the outputs (feature maps) are passed to a SVM for classification. The regression between predicted bounding boxes (bboxes) and ground-truth bboxes are computed.

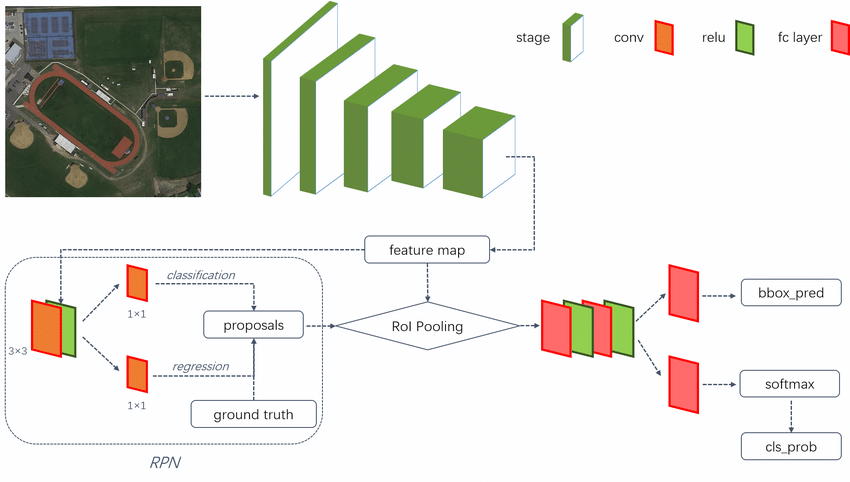
**Faster R-CNN**(frcnn for short) makes further progress than Fast R-CNN. Search selective process is replaced by **Region Proposal Network** (RPN). As the name revealed, RPN is a network to propose regions. For instance, after getting the output feature map from a pre-trained model (VGG-16), if the input image has 600x800x3 dimensions, the output feature map would be 37x50x256 dimensions.

Each point in 37x50 is considered as an anchor. We need to define specific ratios and sizes for each anchor. We then apply a region proposal network (RPM) to the features. This basically predicts if an object is present in that region (or not). Hence, we apply a pooling layer and convert all the regions to the same shape. Next, these regions are passed through a fully connected network so that the class label and bounding boxes are predicted. we first compute the region of interest so that the computation time can be reduced. For all the predicted regions, we compute the Intersection over Union (IoU) with the ground truth boxes. We can computer IoU like this:

IoU = Area of the intersection / Area of the union

**Now, only if the IoU is greater than or equal to 0.5, we consider that as a region of interest(RoI). Otherwise, we neglect that particular region. We do this for all the regions and then select only a set of regions for which the IoU is greater than 0.5.**

RPN is finished after going through the above steps. Then we go to the second stage of frcnn. Similar to Fast R-CNN, ROI pooling is used for these proposed regions (ROIs). The output is 7x7x512. Then, we flatten this layer with some fully connected layers. The final step is a softmax function for classification and linear regression to fix the boxes’ location.



Given below are the parameters we’ve defined

# Faster R-CNN with Inception v2, configured for Oxford-IIIT Pets Dataset.

# Users should configure the fine\_tune\_checkpoint field in the train config as

# well as the label\_map\_path and input\_path fields in the train\_input\_reader and

# eval\_input\_reader. Search for "PATH\_TO\_BE\_CONFIGURED" to find the fields that

# should be configured.

model {

faster\_rcnn {

num\_classes: 15

image\_resizer {

keep\_aspect\_ratio\_resizer {

min\_dimension: 600

max\_dimension: 1024

}

}

feature\_extractor {

type: 'faster\_rcnn\_inception\_v2'

first\_stage\_features\_stride: 16

}

first\_stage\_anchor\_generator {

grid\_anchor\_generator {

scales: [0.25, 0.5, 1.0, 2.0]

aspect\_ratios: [0.5, 1.0, 2.0]

height\_stride: 16

width\_stride: 16

}

}

first\_stage\_box\_predictor\_conv\_hyperparams {

op: CONV

regularizer {

l2\_regularizer {

weight: 0.0

}

}

initializer {

truncated\_normal\_initializer {

stddev: 0.01

}

}

}

first\_stage\_nms\_score\_threshold: 0.0

first\_stage\_nms\_iou\_threshold: 0.7

first\_stage\_max\_proposals: 300

first\_stage\_localization\_loss\_weight: 2.0

first\_stage\_objectness\_loss\_weight: 1.0

initial\_crop\_size: 14

maxpool\_kernel\_size: 2

maxpool\_stride: 2

second\_stage\_box\_predictor {

mask\_rcnn\_box\_predictor {

use\_dropout: false

dropout\_keep\_probability: 1.0

fc\_hyperparams {

op: FC

regularizer {

l2\_regularizer {

weight: 0.0

}

}

initializer {

variance\_scaling\_initializer {

factor: 1.0

uniform: true

mode: FAN\_AVG

}

}

}

}

}

second\_stage\_post\_processing {

batch\_non\_max\_suppression {

score\_threshold: 0.0

iou\_threshold: 0.6

max\_detections\_per\_class: 100

max\_total\_detections: 300

}

score\_converter: SOFTMAX

}

second\_stage\_localization\_loss\_weight: 2.0

second\_stage\_classification\_loss\_weight: 1.0

}

}

train\_config: {

batch\_size: 12

optimizer {

momentum\_optimizer: {

learning\_rate: {

manual\_step\_learning\_rate {

initial\_learning\_rate: 0.0002

schedule {

step: 900000

learning\_rate: .00002

}

schedule {

step: 1200000

learning\_rate: .000002

}

}

}

momentum\_optimizer\_value: 0.9

}

use\_moving\_average: false

}

gradient\_clipping\_by\_norm: 10.0

fine\_tune\_checkpoint: "/content/drive/MyDrive/cars/Tensorflow/models/research/pretrained\_model/model.ckpt"

from\_detection\_checkpoint: true

load\_all\_detection\_checkpoint\_vars: true

# Note: The below line limits the training process to 200K steps, which we

# empirically found to be sufficient enough to train the pets dataset. This

# effectively bypasses the learning rate schedule (the learning rate will

# never decay). Remove the below line to train indefinitely.

num\_steps: 10000

data\_augmentation\_options {

random\_horizontal\_flip {

}

}

}

train\_input\_reader: {

tf\_record\_input\_reader {

input\_path: "/content/drive/MyDrive/cars/Tensorflow/workspace/training\_demo/annotations/test/test.tfrecord"

}

label\_map\_path: "/content/drive/MyDrive/cars/Tensorflow/workspace/training\_demo/annotations/train/label\_map.pbtxt"

}

eval\_config: {

metrics\_set: "coco\_detection\_metrics"

num\_examples: 1101

}

eval\_input\_reader: {

tf\_record\_input\_reader {

input\_path: "/content/drive/MyDrive/cars/Tensorflow/workspace/training\_demo/annotations/train/train.tfrecord"

}

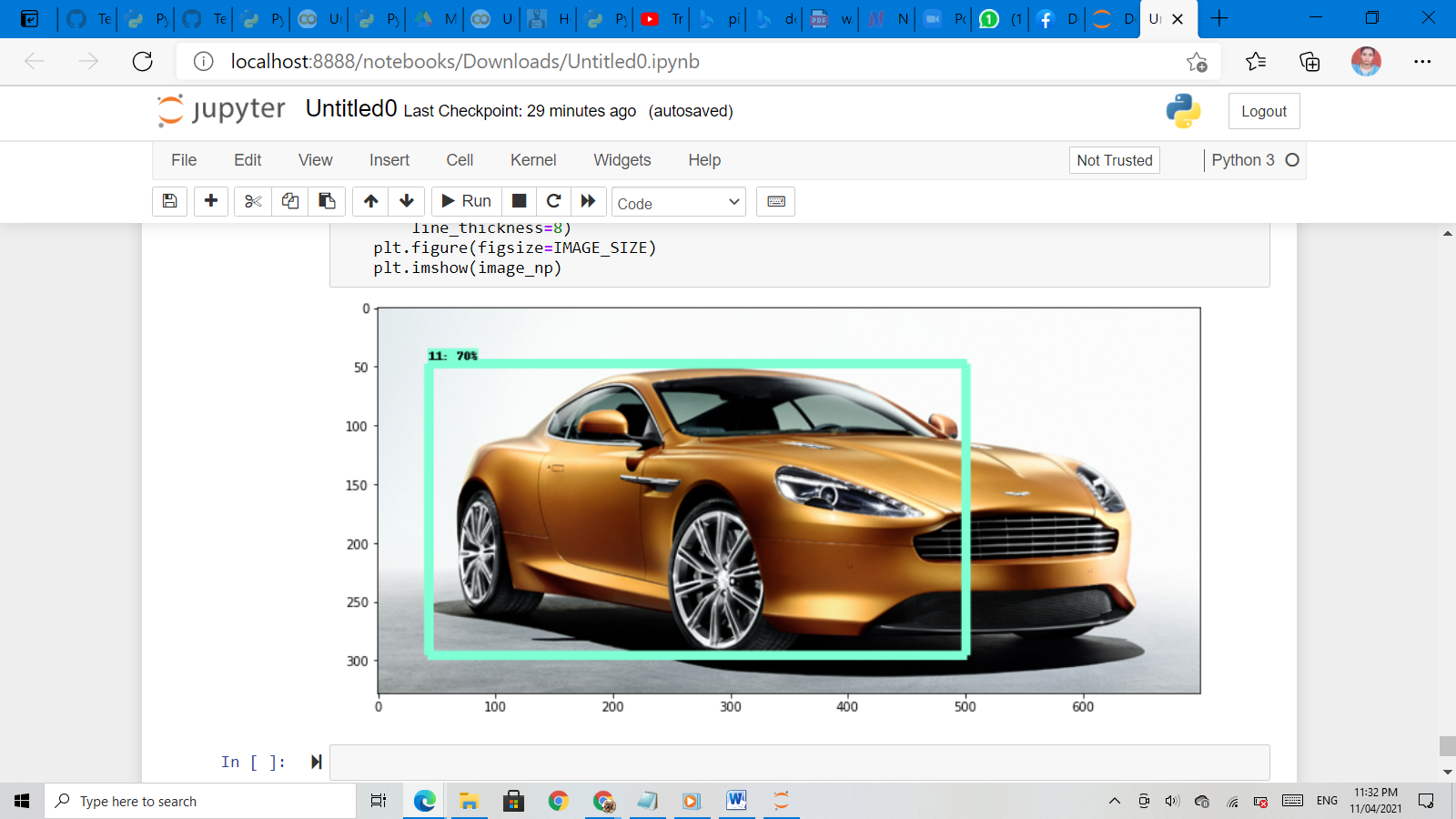
label\_map\_path: "/content/drive/MyDrive/cars/Tensorflow/workspace/training\_demo/annotations/train/label\_map.pbtxt"

shuffle: false

num\_readers: 1

}

After training our model, we get Loss for final step: 0.5709841.



We can see that our accuracy is around 70%.

MobileNet

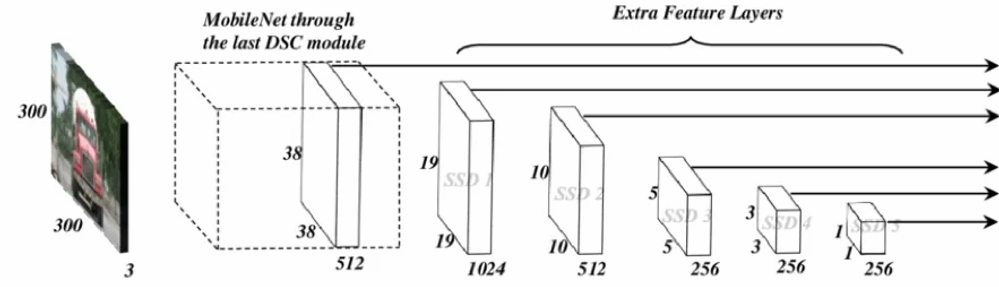
MobileNet is a CNN architecture model for Image Classification and Mobile Vision. There are other models as well but what makes MobileNet special that it very less computation power to run or apply transfer learning to. This makes it a perfect fit for Mobile devices, embedded systems and computers without GPU or low computational efficiency with compromising significantly with the accuracy of the results. It is also best suited for web browsers as browsers have limitation over computation, graphic processing and storage.

MobileNets for mobile and embedded vision applications is proposed, which are based on a streamlined architecture that uses depthwise separable convolutions to build light weight deep neural networks.

Two simple global hyper-parameters that efficiently trade off between latency and accuracy are introduced.

The core layer of MobileNet is depthwise separable filters, named as Depthwise Separable Convolution. The network structure is another factor to boost the performance. Finally, the width and resolution can be tuned to trade off between latency and accuracy.

Below we have the MobileNet architecture

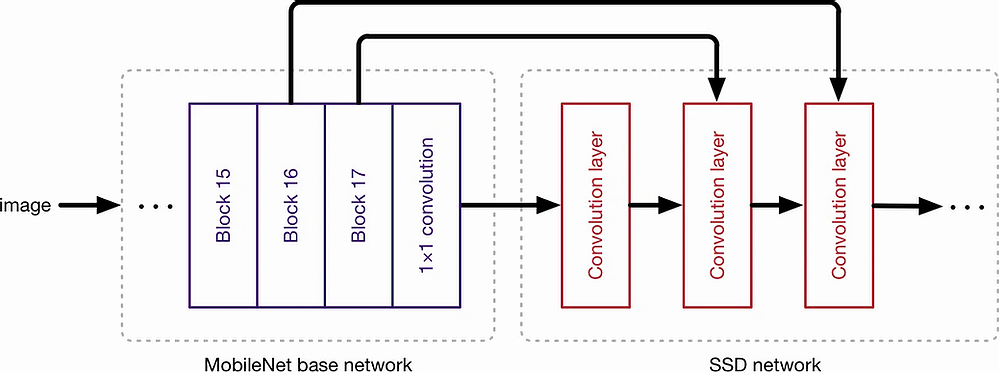


It’s composed of two parts:

1. Extract feature maps, and

2. Apply convolution filter to detect objects

To further tackle the practical limitations of running high resource and power-consuming neural networks on low-end devices in real-time applications, MobileNet was integrated into the SSD framework. So, when MobileNet is used as the base network in the SSD, it became **MobileNet SSD.**



The model and features defined are as follows

# SSD with Mobilenet v2 configuration for MSCOCO Dataset.

# Users should configure the fine\_tune\_checkpoint field in the train config as

# well as the label\_map\_path and input\_path fields in the train\_input\_reader and

# eval\_input\_reader. Search for "PATH\_TO\_BE\_CONFIGURED" to find the fields that

# should be configured.

model {

ssd {

num\_classes: 1

box\_coder {

faster\_rcnn\_box\_coder {

y\_scale: 10.0

x\_scale: 10.0

height\_scale: 5.0

width\_scale: 5.0

}

}

matcher {

argmax\_matcher {

matched\_threshold: 0.5

unmatched\_threshold: 0.5

ignore\_thresholds: false

negatives\_lower\_than\_unmatched: true

force\_match\_for\_each\_row: true

}

}

similarity\_calculator {

iou\_similarity {

}

}

anchor\_generator {

ssd\_anchor\_generator {

num\_layers: 6

min\_scale: 0.2

max\_scale: 0.95

aspect\_ratios: 1.0

aspect\_ratios: 2.0

aspect\_ratios: 0.5

aspect\_ratios: 3.0

aspect\_ratios: 0.3333

}

}

image\_resizer {

fixed\_shape\_resizer {

height: 300

width: 300

}

}

box\_predictor {

convolutional\_box\_predictor {

min\_depth: 0

max\_depth: 0

num\_layers\_before\_predictor: 0

use\_dropout: false

dropout\_keep\_probability: 0.8

kernel\_size: 1

box\_code\_size: 4

apply\_sigmoid\_to\_scores: false

conv\_hyperparams {

activation: RELU\_6,

regularizer {

l2\_regularizer {

weight: 0.00004

}

}

initializer {

truncated\_normal\_initializer {

stddev: 0.03

mean: 0.0

}

}

batch\_norm {

train: true,

scale: true,

center: true,

decay: 0.9997,

epsilon: 0.001,

}

}

}

}

feature\_extractor {

type: 'ssd\_mobilenet\_v2'

min\_depth: 16

depth\_multiplier: 1.0

conv\_hyperparams {

activation: RELU\_6,

regularizer {

l2\_regularizer {

weight: 0.00004

}

}

initializer {

truncated\_normal\_initializer {

stddev: 0.03

mean: 0.0

}

}

batch\_norm {

train: true,

scale: true,

center: true,

decay: 0.9997,

epsilon: 0.001,

}

}

}

loss {

classification\_loss {

weighted\_sigmoid {

}

}

localization\_loss {

weighted\_smooth\_l1 {

}

}

hard\_example\_miner {

num\_hard\_examples: 3000

iou\_threshold: 0.99

loss\_type: CLASSIFICATION

max\_negatives\_per\_positive: 3

min\_negatives\_per\_image: 3

}

classification\_weight: 1.0

localization\_weight: 1.0

}

normalize\_loss\_by\_num\_matches: true

post\_processing {

batch\_non\_max\_suppression {

score\_threshold: 1e-8

iou\_threshold: 0.6

max\_detections\_per\_class: 100

max\_total\_detections: 100

}

score\_converter: SIGMOID

}

}

}

train\_config: {

batch\_size: 12

optimizer {

rms\_prop\_optimizer: {

learning\_rate: {

exponential\_decay\_learning\_rate {

initial\_learning\_rate: 0.004

decay\_steps: 800720

decay\_factor: 0.95

}

}

momentum\_optimizer\_value: 0.9

decay: 0.9

epsilon: 1.0

}

}

fine\_tune\_checkpoint: "/content/models/research/pretrained\_model/model.ckpt"

fine\_tune\_checkpoint\_type: "detection"

# Note: The below line limits the training process to 200K steps, which we

# empirically found to be sufficient enough to train the pets dataset. This

# effectively bypasses the learning rate schedule (the learning rate will

# never decay). Remove the below line to train indefinitely.

num\_steps: 30000

data\_augmentation\_options {

random\_horizontal\_flip {

}

}

data\_augmentation\_options {

ssd\_random\_crop {

}

}

}

train\_input\_reader: {

tf\_record\_input\_reader {

input\_path: "/content/drive/MyDrive/gl/cars\_train.tfrecord"

}

label\_map\_path: "/content/drive/MyDrive/gl/cars\_label\_map.pbtxt"

}

eval\_config: {

num\_examples: 8000

# Note: The below line limits the evaluation process to 10 evaluations.

# Remove the below line to evaluate indefinitely.

max\_evals: 10

}

eval\_input\_reader: {

tf\_record\_input\_reader {

input\_path: "/content/drive/MyDrive/gl/cars\_test.tfrecord"

}

label\_map\_path: "/content/drive/MyDrive/gl/cars\_label\_map.pbtxt"

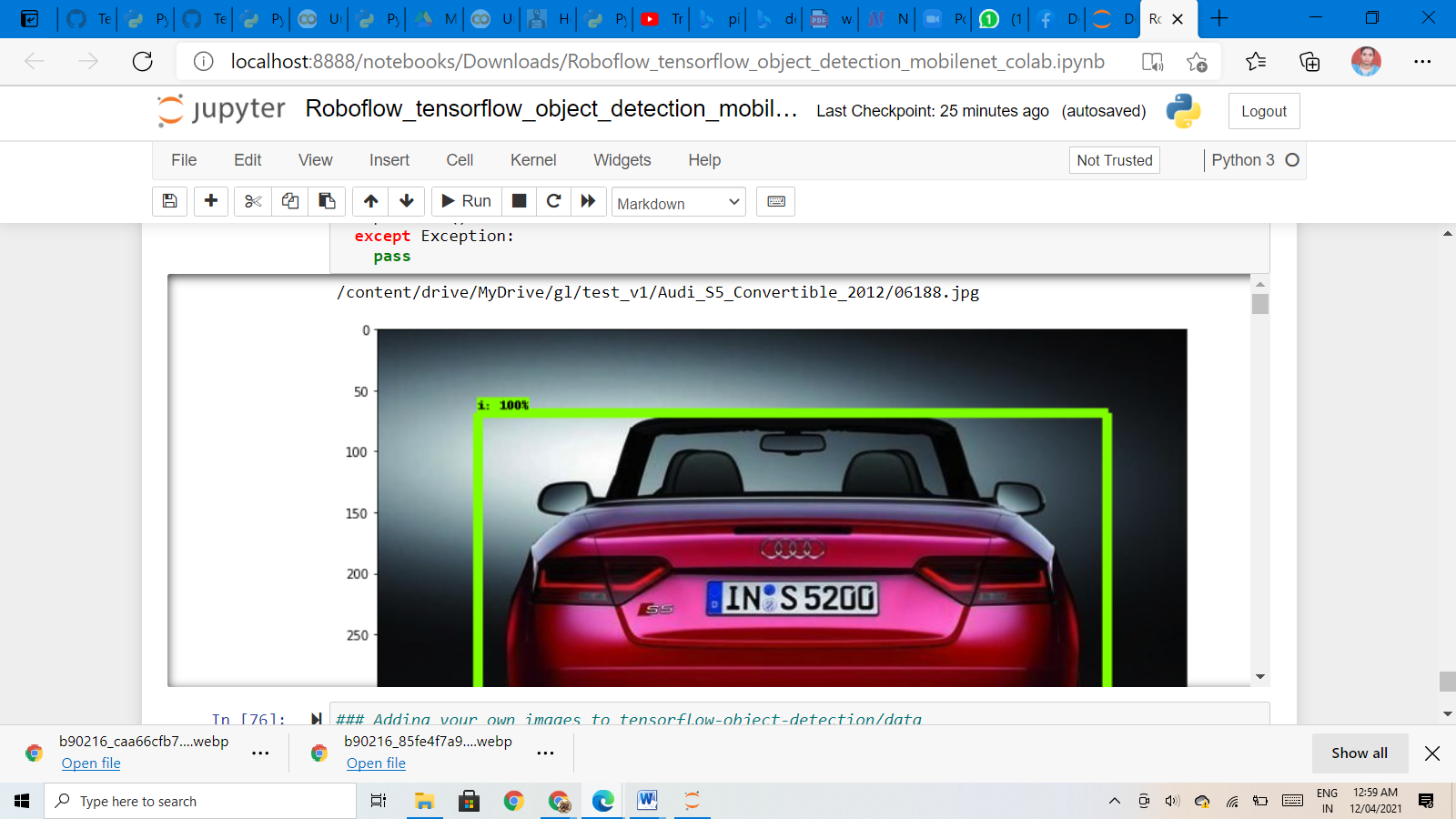
shuffle: false

num\_readers: 1

}

After training our model we find Loss for final step: 0.5913437. This is slightly higher than the faster RCNN model

The final test output:



INFFERENCE

While both he models perform well, MobileNet performed better in terms of speed on our machine. However, the accuracy of the Faster R-CNN model was better than the SSD MobileNet model.

Taking advantage of the considerably smaller size of MobileNet v2, this model performed better as compared to Faster R-CNN in terms of speed especially on videos yielding more Frames per second on our test machine. Hence, MobileNet v2 can be used in real time object detection. However, with more training data, both models can perform considerably better.

Hence depending on the design considerations the model can be chosen, but for our project we’ve chosen Faster R-CNN due to its better accuracy.

Building IU Model

User interface (UI) is everything designed into an information device with which a person may interact. This can include display screens, keyboards, a mouse and the appearance of a desktop. It is also the way through which a user interacts with an application or a website.

We’ve designed an UI for the two modules ie, 1)training and testing, 2)image classification and object detection.

We’ve used Django and Flask frameworks for the task of creating a web application and mobile app.

Django is a Python-based web framework that allows you to quickly create efficient web applications. It is also called batteries included framework because Django provides built-in features for everything including Django Admin Interface, default database – SQLlite3, etc. When you’re building a website, you always need a similar set of components: a way to handle user authentication (signing up, signing in, signing out), a management panel for your website, forms, a way to upload files, etc. Django gives you ready-made components to use and that too for rapid development.

Flask is a web application framework written in Python. Armin Ronacher, who leads an international group of Python enthusiasts named Pocco, developed it. It is classified as a microframework because it does not require particular tools or libraries.[2] It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Extensions exist for object-relational mappers, form validation, upload handling, various open authentication technologies and several common framework related tools.

Process

To use Django Models, one needs to have a project and an app working in it. After you start an app you can create models in app/models.py. Django maps the fields defined in Django models into table fields of the database.

Whenever we create a Model, Delete a Model, or update anything in any of models.py of our project. We need to run two commands makemigrations and migrate. makemigrations basically generates the SQL commands for preinstalled apps (which can be viewed in installed apps in settings.py) and your newly created app’s model which you add in installed apps whereas migrate executes those SQL commands in the database file.

Django lets us interact with its database models, i.e. add, delete, modify and query objects, using a database-abstraction API called ORM(Object Relational Mapper).

To launch the application

Initially we copy the requirement file into the project folder.

Activate the environment.

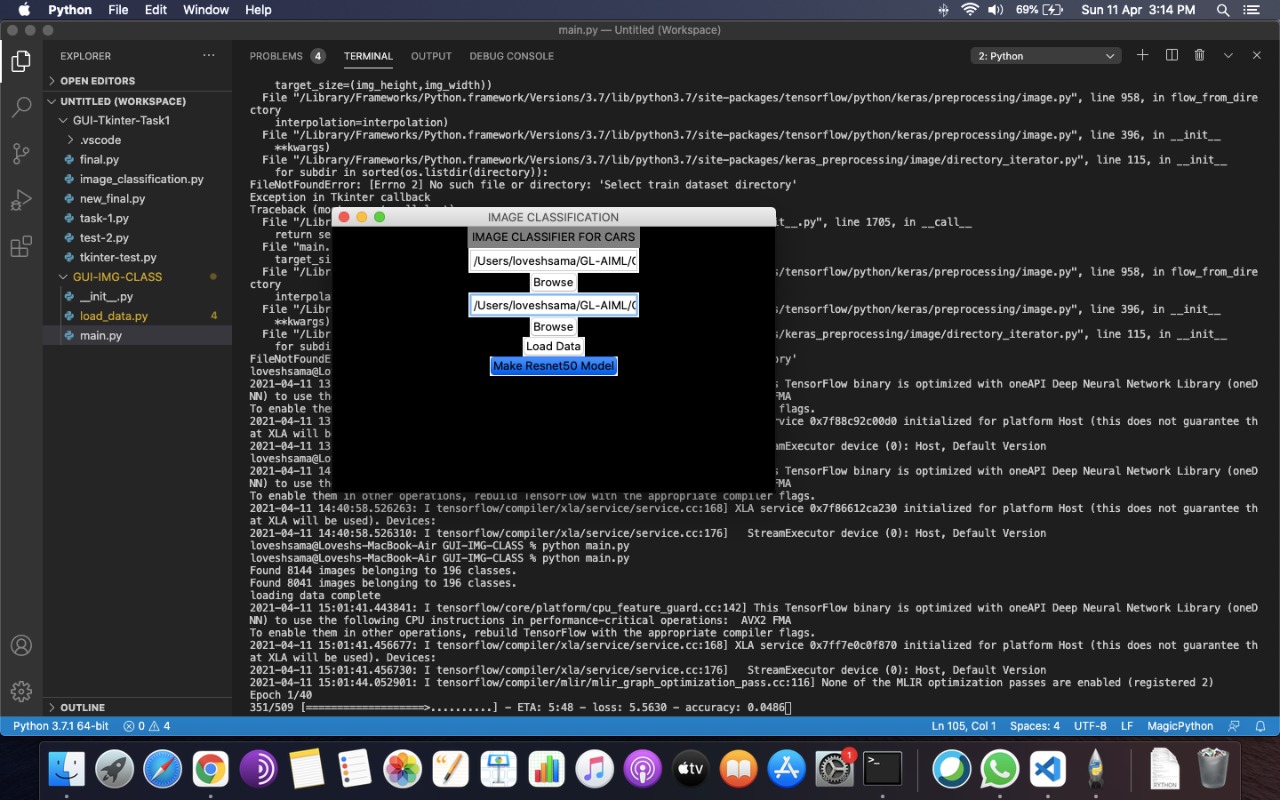
Run python install requirements.txt.

Run the application using python manage.py runserver.

Outputs for Module 1: Train and test data



Outputs for Module 2: Object classification



Outputs for Module 2: Object Detection

We load the custom trained weights using load\_weights.py script. This will convert the yolov3 weights into TensorFlow .ckpt model files. Then run a Flask application to create two object detections APIs in order to get detections through REST endpoints. Initialize and run the Flask app on port 5000 of your local machine. While app.py is running the first available API is a POST routed to /detections on port 5000 of localhost. This endpoint takes in images as input and returns a JSON response with all the detections found within each image (classes found within the images and the associated confidence). The APIs are tested using Postman or through Curl commands.

The uploaded image is returned with detections drawn.

Note: Object detection uses API and files which are environment dependent hence can’t be automated