TRAFFIC MANAGEMENT SYSTEM

ABSTRACT:

In today's rapidly urbanizing world, traffic congestion is a pervasive problem that affects millions of people daily. This study proposes an innovative approach to address this issue by leveraging historical traffic data and advanced machine learning algorithms. By analyzing and modeling historical traffic patterns, we aim to predict and understand congestion dynamics, ultimately leading to more effective traffic management and improved urban mobility.

This research begins by collecting extensive historical traffic data, including information on traffic volume, weather conditions, time of day, and road infrastructure. These datasets serve as the foundation for training and validating machine learning models. Through the application of state-of-the-art algorithms such as deep neural networks and recurrent neural networks (RNNs), we extract meaningful insights from the data.

The predictive models developed in this study will not only forecast traffic congestion but also identify key contributing factors and potential hotspots. These insights can guide policymakers, urban planners, and transportation authorities in making informed decisions to mitigate congestion and enhance the overall quality of life for residents.

Our findings promise to revolutionize the way we approach traffic management, moving towards a more proactive and data-driven strategy. By harnessing the power of historical data and machine learning, we aim to create a future where traffic congestion becomes a more manageable challenge, ultimately leading to more efficient and sustainable urban environments.

ALGORITHM:

1. **Data Collection:**

- Gather historical traffic data, including information such as traffic volume, speed, weather conditions, and time of day.
- Ensure data quality by cleaning and preprocessing it. Handle missing values and outliers appropriately.

2. **Feature Engineering:**

   - Extract relevant features from the data that can influence traffic congestion, such as:
     - Day of the week
     - Time of day (rush hours, non-peak hours)
     - Weather conditions (rain, snow, temperature)
     - Special events or holidays
     - Road infrastructure data (number of lanes, road type)
   - Create additional features like historical traffic patterns and trends.

3. **Data Splitting:**

   - Divide the dataset into training, validation, and testing sets to evaluate the model's performance effectively.

4. **Model Selection:**

   - Choose appropriate machine learning algorithms for traffic prediction. Common choices include:
     - Regression models (e.g., Linear Regression)
     - Time-series forecasting models (e.g., ARIMA, LSTM)
     - Ensemble methods (e.g., Random Forest, Gradient Boosting)
     - Neural networks (e.g., CNN or RNN-based models)

5. **Model Training:**

   - Train the selected machine learning models using the training dataset.
   - Tune hyperparameters and optimize the model's performance.

6. **Model Evaluation:**

   - Evaluate the model's performance using the validation dataset, using metrics like Mean Absolute Error (MAE), Mean Squared Error (MSE), or Root Mean Squared Error (RMSE).

7. **Model Testing:**

   - Test the trained model on the testing dataset to assess its real-world performance and generalization.

8. **Prediction:**

   - Utilize the trained model to make real-time or future traffic congestion predictions.
   - Monitor and update the model periodically with new data to ensure it remains accurate.

9. **Deployment:**

   - Integrate the predictive model into a traffic management system or a mobile application for users to access congestion predictions.

10. **Feedback Loop:**

    - Continuously collect new traffic data to keep the model up-to-date and refine its predictions over time.
    - Re-train the model periodically to adapt to changing traffic patterns.

Remember that the effectiveness of this algorithm depends on the quality and quantity of historical traffic data, the choice of appropriate features, and the selection and tuning of machine learning models. Regular updates and feedback are crucial to maintaining accurate congestion

PROGRAM:

```
# NO. OF VEHICLES IN SIGNAL CLASS
# stops not used
# DISTRIBUTION
# BUS TOUCHING ON TURNS
# Distribution using python class

# *** IMAGE XY COOD IS TOP LEFT
Import random
Import math
Import time
Import threading
# from vehicle_detection import detection
Import pygame
Import sys
Import os

# options={
#    'model':'./cfg/yolo.cfg',     #specifying the path of model
#    'load':'./bin/yolov2.weights',   #weights
#    'threshold':0.3     #minimum confidence factor to create a box, greater than 0.3 good
# }

# tfnet=TFNet(options)    #READ ABOUT TFNET

# Default values of signal times
defaultRed = 150
defaultYellow = 5
defaultGreen = 20
defaultMinimum = 10
defaultMaximum = 60

signals = []
```

```
noOfSignals = 4
simTime = 300       # change this to change time of simulation
timeElapsed = 0

currentGreen = 0   # Indicates which signal is green
nextGreen = (currentGreen+1)%noOfSignals
currentYellow = 0   # Indicates whether yellow signal is on or off

# Average times for vehicles to pass the intersection
carTime = 2
bikeTime = 1
rickshawTime = 2.25
busTime = 2.5
truckTime = 2.5

# Count of cars at a traffic signal
noOfCars = 0
noOfBikes = 0
noOfBuses =0
noOfTrucks = 0
noOfRickshaws = 0
noOfLanes = 2

# Red signal time at which cars will be detected at a signal
detectionTime = 5

speeds = {'car':2.25, 'bus':1.8, 'truck':1.8, 'rickshaw':2, 'bike':2.5}  # average speeds of vehicles

# Coordinates of start
X = {'right':[0,0,0], 'down':[755,727,697], 'left':[1400,1400,1400], 'up':[602,627,657]}
Y = {'right':[348,370,398], 'down':[0,0,0], 'left':[498,466,436], 'up':[800,800,800]}

Vehicles = {'right': {0:[], 1:[], 2:[], 'crossed':0}, 'down': {0:[], 1:[], 2:[], 'crossed':0}, 'left': {0:[], 1:[],
2:[], 'crossed':0}, 'up': {0:[], 1:[], 2:[], 'crossed':0}}
```

```
vehicleTypes = {0:’car’, 1:’bus’, 2:’truck’, 3:’rickshaw’, 4:’bike’}
directionNumbers = {0:’right’, 1:’down’, 2:’left’, 3:’up’}

# Coordinates of signal image, timer, and vehicle count
signalCoods = [(530,230),(810,230),(810,570),(530,570)]
signalTimerCoods = [(530,210),(810,210),(810,550),(530,550)]
vehicleCountCoods = [(480,210),(880,210),(880,550),(480,550)]
vehicleCountTexts = [“0”, “0”, “0”, “0”]

# Coordinates of stop lines
stopLines = {‘right’: 590, ‘down’: 330, ‘left’: 800, ‘up’: 535}
defaultStop = {‘right’: 580, ‘down’: 320, ‘left’: 810, ‘up’: 545}
stops = {‘right’: [580,580,580], ‘down’: [320,320,320], ‘left’: [810,810,810], ‘up’: [545,545,545]}

mid = {‘right’: {‘x’:705, ‘y’:445}, ‘down’: {‘x’:695, ‘y’:450}, ‘left’: {‘x’:695, ‘y’:425}, ‘up’: {‘x’:695, ‘y’:400}}
rotationAngle = 3

# Gap between vehicles
Gap = 15    # stopping gap
Gap2 = 15   # moving gap

Pygame.init()
Simulation = pygame.sprite.Group()

Class TrafficSignal:
    Def __init__(self, red, yellow, green, minimum, maximum):
        Self.red = red
        Self.yellow = yellow
        Self.green = green
        Self.minimum = minimum
        Self.maximum = maximum
        Self.signalText = “30”
        Self.totalGreenTime = 0
```

```
Class Vehicle(pygame.sprite.Sprite):
    Def __init__(self, lane, vehicleClass, direction_number, direction, will_turn):
        Pygame.sprite.Sprite.__init__(self)
        Self.lane = lane
        Self.vehicleClass = vehicleClass
        Self.speed = speeds[vehicleClass]
        Self.direction_number = direction_number
        Self.direction = direction
        Self.x = x[direction][lane]
        Self.y = y[direction][lane]
        Self.crossed = 0
        Self.willTurn = will_turn
        Self.turned = 0
        Self.rotateAngle = 0
        Vehicles[direction][lane].append(self)
        # self.stop = stops[direction][lane]
        Self.index = len(vehicles[direction][lane]) – 1
        Path = “images/” + direction + “/” + vehicleClass + “.png”
        Self.originalImage = pygame.image.load(path)
        Self.currentImage = pygame.image.load(path)



        If(direction==’right’):
            If(len(vehicles[direction][lane])>1 and vehicles[direction][lane][self.index-1].crossed==0):
# if more than 1 vehicle in the lane of vehicle before it has crossed stop line
                Self.stop = vehicles[direction][lane][self.index-1].stop –
vehicles[direction][lane][self.index-1].currentImage.get_rect().width – gap          # setting stop
coordinate as: stop coordinate of next vehicle – width of next vehicle – gap
            Else:
                Self.stop = defaultStop[direction]
            # Set new starting and stopping coordinate
            Temp = self.currentImage.get_rect().width + gap
            X[direction][lane] -= temp
```

```
            Stops[direction][lane] -= temp
        Elif(direction==’left’):
            If(len(vehicles[direction][lane])>1 and vehicles[direction][lane][self.index-1].crossed==0):
                Self.stop = vehicles[direction][lane][self.index-1].stop +
vehicles[direction][lane][self.index-1].currentImage.get_rect().width + gap
            Else:
                Self.stop = defaultStop[direction]
            Temp = self.currentImage.get_rect().width + gap
            X[direction][lane] += temp
            Stops[direction][lane] += temp
        Elif(direction==’down’):
            If(len(vehicles[direction][lane])>1 and vehicles[direction][lane][self.index-1].crossed==0):
                Self.stop = vehicles[direction][lane][self.index-1].stop –
vehicles[direction][lane][self.index-1].currentImage.get_rect().height – gap
            Else:
                Self.stop = defaultStop[direction]
            Temp = self.currentImage.get_rect().height + gap
            Y[direction][lane] -= temp
            Stops[direction][lane] -= temp
        Elif(direction==’up’):
            If(len(vehicles[direction][lane])>1 and vehicles[direction][lane][self.index-1].crossed==0):
                Self.stop = vehicles[direction][lane][self.index-1].stop +
vehicles[direction][lane][self.index-1].currentImage.get_rect().height + gap
            Else:
                Self.stop = defaultStop[direction]
            Temp = self.currentImage.get_rect().height + gap
            Y[direction][lane] += temp
            Stops[direction][lane] += temp
        Simulation.add(self)

    Def render(self, screen):
        Screen.blit(self.currentImage, (self.x, self.y))

    Def move(self):
```

```
        If(self.direction==’right’):
            If(self.crossed==0 and
self.x+self.currentImage.get_rect().width>stopLines[self.direction]):   # if the image has crossed
stop line now
                Self.crossed = 1
                Vehicles[self.direction][‘crossed’] += 1
            If(self.willTurn==1):
                If(self.crossed==0 or self.x+self.currentImage.get_rect().width<mid[self.direction][‘x’]):
                    If((self.x+self.currentImage.get_rect().width<=self.stop or (currentGreen==0 and
currentYellow==0) or self.crossed==1) and (self.index==0 or
self.x+self.currentImage.get_rect().width<(vehicles[self.direction][self.lane][self.index-1].x –
gap2) or vehicles[self.direction][self.lane][self.index-1].turned==1)):
                        Self.x += self.speed
                Else:
                    If(self.turned==0):
                        Self.rotateAngle += rotationAngle
                        Self.currentImage = pygame.transform.rotate(self.originalImage, -
self.rotateAngle)
                        Self.x += 2
                        Self.y += 1.8
                        If(self.rotateAngle==90):
                            Self.turned = 1
                            # path = “images/” +
directionNumbers[((self.direction_number+1)%noOfSignals)] + “/” + self.vehicleClass + “.png”
                            # self.x = mid[self.direction][‘x’]
                            # self.y = mid[self.direction][‘y’]
                            # self.image = pygame.image.load(path)
                    Else:
                        If(self.index==0 or
self.y+self.currentImage.get_rect().height<(vehicles[self.direction][self.lane][self.index-1].y –
gap2) or self.x+self.currentImage.get_rect().width<(vehicles[self.direction][self.lane][self.index-
1].x – gap2)):
                            Self.y += self.speed
            Else:
                If((self.x+self.currentImage.get_rect().width<=self.stop or self.crossed == 1 or
(currentGreen==0 and currentYellow==0)) and (self.index==0 or
self.x+self.currentImage.get_rect().width<(vehicles[self.direction][self.lane][self.index-1].x –
gap2) or (vehicles[self.direction][self.lane][self.index-1].turned==1))):
```

```
            # (if the image has not reached its stop coordinate or has crossed stop line or has green signal) and (it is either the first vehicle in that lane or it is has enough gap to the next vehicle in that lane)
                Self.x += self.speed  # move the vehicle




        Elif(self.direction==’down’):
            If(self.crossed==0 and self.y+self.currentImage.get_rect().height>stopLines[self.direction]):
                Self.crossed = 1
                Vehicles[self.direction][‘crossed’] += 1
            If(self.willTurn==1):
                If(self.crossed==0 or self.y+self.currentImage.get_rect().height<mid[self.direction][‘y’]):
                    If((self.y+self.currentImage.get_rect().height<=self.stop or (currentGreen==1 and currentYellow==0) or self.crossed==1) and (self.index==0 or self.y+self.currentImage.get_rect().height<(vehicles[self.direction][self.lane][self.index-1].y – gap2) or vehicles[self.direction][self.lane][self.index-1].turned==1)):
                        Self.y += self.speed
                Else:
                    If(self.turned==0):
                        Self.rotateAngle += rotationAngle
                        Self.currentImage = pygame.transform.rotate(self.originalImage, -self.rotateAngle)
                        Self.x -= 2.5
                        Self.y += 2
                        If(self.rotateAngle==90):
                            Self.turned = 1
                    Else:
                        If(self.index==0 or self.x>(vehicles[self.direction][self.lane][self.index-1].x + vehicles[self.direction][self.lane][self.index-1].currentImage.get_rect().width + gap2) or self.y<(vehicles[self.direction][self.lane][self.index-1].y – gap2)):
                            Self.x -= self.speed
            Else:
                If((self.y+self.currentImage.get_rect().height<=self.stop or self.crossed == 1 or (currentGreen==1 and currentYellow==0)) and (self.index==0 or self.y+self.currentImage.get_rect().height<(vehicles[self.direction][self.lane][self.index-1].y – gap2) or (vehicles[self.direction][self.lane][self.index-1].turned==1))):
```

```
                Self.y += self.speed


        Elif(self.direction==’left’):
            If(self.crossed==0 and self.x<stopLines[self.direction]):
                Self.crossed = 1
                Vehicles[self.direction][‘crossed’] += 1
            If(self.willTurn==1):
                If(self.crossed==0 or self.x>mid[self.direction][‘x’]):
                    If((self.x>=self.stop or (currentGreen==2 and currentYellow==0) or
self.crossed==1) and (self.index==0 or self.x>(vehicles[self.direction][self.lane][self.index-1].x +
vehicles[self.direction][self.lane][self.index-1].currentImage.get_rect().width + gap2) or
vehicles[self.direction][self.lane][self.index-1].turned==1)):
                        Self.x -= self.speed
                Else:
                    If(self.turned==0):
                        Self.rotateAngle += rotationAngle
                        Self.currentImage = pygame.transform.rotate(self.originalImage, -
self.rotateAngle)
                        Self.x -= 1.8
                        Self.y -= 2.5
                        If(self.rotateAngle==90):
                            Self.turned = 1
                            # path = “images/” +
directionNumbers[((self.direction_number+1)%noOfSignals)] + “/” + self.vehicleClass + “.png”
                            # self.x = mid[self.direction][‘x’]
                            # self.y = mid[self.direction][‘y’]
                            # self.currentImage = pygame.image.load(path)
                    Else:
                        If(self.index==0 or self.y>(vehicles[self.direction][self.lane][self.index-1].y +
vehicles[self.direction][self.lane][self.index-1].currentImage.get_rect().height +  gap2) or
self.x>(vehicles[self.direction][self.lane][self.index-1].x + gap2)):
                            Self.y -= self.speed
            Else:
                If((self.x>=self.stop or self.crossed == 1 or (currentGreen==2 and currentYellow==0))
and (self.index==0 or self.x>(vehicles[self.direction][self.lane][self.index-1].x +
vehicles[self.direction][self.lane][self.index-1].currentImage.get_rect().width + gap2) or
(vehicles[self.direction][self.lane][self.index-1].turned==1))):
```

```
            # (if the image has not reached its stop coordinate or has crossed stop line or has green signal) and (it is either the first vehicle in that lane or it is has enough gap to the next vehicle in that lane)
                Self.x -= self.speed  # move the vehicle
        # if((self.x>=self.stop or self.crossed == 1 or (currentGreen==2 and currentYellow==0)) and (self.index==0 or self.x>(vehicles[self.direction][self.lane][self.index-1].x + vehicles[self.direction][self.lane][self.index-1].currentImage.get_rect().width + gap2))):
        #     self.x -= self.speed
    Elif(self.direction==’up’):
        If(self.crossed==0 and self.y<stopLines[self.direction]):
            Self.crossed = 1
            Vehicles[self.direction][‘crossed’] += 1
        If(self.willTurn==1):
            If(self.crossed==0 or self.y>mid[self.direction][‘y’]):
                If((self.y>=self.stop or (currentGreen==3 and currentYellow==0) or self.crossed == 1) and (self.index==0 or self.y>(vehicles[self.direction][self.lane][self.index-1].y + vehicles[self.direction][self.lane][self.index-1].currentImage.get_rect().height +  gap2) or vehicles[self.direction][self.lane][self.index-1].turned==1)):
                    Self.y -= self.speed
            Else:
                If(self.turned==0):
                    Self.rotateAngle += rotationAngle
                    Self.currentImage = pygame.transform.rotate(self.originalImage, -self.rotateAngle)
                    Self.x += 1
                    Self.y -= 1
                    If(self.rotateAngle==90):
                        Self.turned = 1
                Else:
                    If(self.index==0 or self.x<(vehicles[self.direction][self.lane][self.index-1].x – vehicles[self.direction][self.lane][self.index-1].currentImage.get_rect().width – gap2) or self.y>(vehicles[self.direction][self.lane][self.index-1].y + gap2)):
                        Self.x += self.speed
        Else:
            If((self.y>=self.stop or self.crossed == 1 or (currentGreen==3 and currentYellow==0)) and (self.index==0 or self.y>(vehicles[self.direction][self.lane][self.index-1].y + vehicles[self.direction][self.lane][self.index-1].currentImage.get_rect().height + gap2) or (vehicles[self.direction][self.lane][self.index-1].turned==1))):
```

```
                Self.y -= self.speed


# Initialization of signals with default values
Def initialize():
    Ts1 = TrafficSignal(0, defaultYellow, defaultGreen, defaultMinimum, defaultMaximum)
    Signals.append(ts1)
    Ts2 = TrafficSignal(ts1.red+ts1.yellow+ts1.green, defaultYellow, defaultGreen,
defaultMinimum, defaultMaximum)
    Signals.append(ts2)
    Ts3 = TrafficSignal(defaultRed, defaultYellow, defaultGreen, defaultMinimum,
defaultMaximum)
    Signals.append(ts3)
    Ts4 = TrafficSignal(defaultRed, defaultYellow, defaultGreen, defaultMinimum,
defaultMaximum)
    Signals.append(ts4)
    Repeat()


# Set time according to formula
Def setTime():
    Global noOfCars, noOfBikes, noOfBuses, noOfTrucks, noOfRickshaws, noOfLanes
    Global carTime, busTime, truckTime, rickshawTime, bikeTime
    Os.system("say detecting vehicles, "+directionNumbers[(currentGreen+1)%noOfSignals])
#    detection_result=detection(currentGreen,tfnet)
#    greenTime = math.ceil(((noOfCars*carTime) + (noOfRickshaws*rickshawTime) +
(noOfBuses*busTime) + (noOfBikes*bikeTime))/(noOfLanes+1))
#    if(greenTime<defaultMinimum):
#        greenTime = defaultMinimum
#    elif(greenTime>defaultMaximum):
#        greenTime = defaultMaximum
    # greenTime =
len(vehicles[currentGreen][0])+len(vehicles[currentGreen][1])+len(vehicles[currentGreen][2])
    # noOfVehicles =
len(vehicles[directionNumbers[nextGreen]][1])+len(vehicles[directionNumbers[nextGreen]][2])-
vehicles[directionNumbers[nextGreen]]['crossed']
    # print("no. of vehicles = ",noOfVehicles)
    noOfCars, noOfBuses, noOfTrucks, noOfRickshaws, noOfBikes = 0,0,0,0,0
```

```
    for j in range(len(vehicles[directionNumbers[nextGreen]][0])):
        vehicle = vehicles[directionNumbers[nextGreen]][0][j]
        if(vehicle.crossed==0):
            vclass = vehicle.vehicleClass
            # print(vclass)
            noOfBikes += 1
    for I in range(1,3):
        for j in range(len(vehicles[directionNumbers[nextGreen]][i])):
            vehicle = vehicles[directionNumbers[nextGreen]][i][j]
            if(vehicle.crossed==0):
                vclass = vehicle.vehicleClass
                # print(vclass)
                If(vclass=='car'):
                    noOfCars += 1
                elif(vclass=='bus'):
                    noOfBuses += 1
                elif(vclass=='truck'):
                    noOfTrucks += 1
                elif(vclass=='rickshaw'):
                    noOfRickshaws += 1
    # print(noOfCars)
    greenTime = math.ceil(((noOfCars*carTime) + (noOfRickshaws*rickshawTime) +
(noOfBuses*busTime) + (noOfTrucks*truckTime)+ (noOfBikes*bikeTime))/(noOfLanes+1))
    # greenTime = math.ceil((noOfVehicles)/noOfLanes)
    Print('Green Time: ',greenTime)
    If(greenTime<defaultMinimum):
        greenTime = defaultMinimum
    elif(greenTime>defaultMaximum):
        greenTime = defaultMaximum
    # greenTime = random.randint(15,50)
    Signals[(currentGreen+1)%(noOfSignals)].green = greenTime


Def repeat():
    Global currentGreen, currentYellow, nextGreen
```

```
    While(signals[currentGreen].green>0):   # while the timer of current green signal is not zero
        printStatus()
        updateValues()
        if(signals[(currentGreen+1)%(noOfSignals)].red==detectionTime):    # set time of next
green signal
            thread = threading.Thread(name="detection",target=setTime, args=())
            thread.daemon = True
            thread.start()
            # setTime()
        Time.sleep(1)
    currentYellow = 1   # set yellow signal on
    vehicleCountTexts[currentGreen] = "0"
    # reset stop coordinates of lanes and vehicles
    For I in range(0,3):
        Stops[directionNumbers[currentGreen]][i] = defaultStop[directionNumbers[currentGreen]]
        For vehicle in vehicles[directionNumbers[currentGreen]][i]:
            Vehicle.stop = defaultStop[directionNumbers[currentGreen]]
    While(signals[currentGreen].yellow>0):  # while the timer of current yellow signal is not zero
        printStatus()
        updateValues()
        time.sleep(1)
    currentYellow = 0   # set yellow signal off

    # reset all signal times of current signal to default times
    Signals[currentGreen].green = defaultGreen
    Signals[currentGreen].yellow = defaultYellow
    Signals[currentGreen].red = defaultRed

    currentGreen = nextGreen # set next signal as green signal
    nextGreen = (currentGreen+1)%noOfSignals    # set next green signal
    signals[nextGreen].red = signals[currentGreen].yellow+signals[currentGreen].green    # set
the red time of next to next signal as (yellow time + green time) of next signal
    repeat()
```

```
# Print the signal timers on cmd
Def printStatus():
        For I in range(0, noOfSignals):
                If(i==currentGreen):
                        If(currentYellow==0):
                                Print(“ GREEN TS”,i+1,”-> r:”,signals[i].red,” y:”,signals[i].yellow,”
g:”,signals[i].green)
                        Else:
                                Print(“YELLOW TS”,i+1,”-> r:”,signals[i].red,” y:”,signals[i].yellow,”
g:”,signals[i].green)
                Else:
                        Print(“   RED TS”,i+1,”-> r:”,signals[i].red,” y:”,signals[i].yellow,”
g:”,signals[i].green)
        Print()

# Update values of the signal timers after every second
Def updateValues():
    For I in range(0, noOfSignals):
        If(i==currentGreen):
            If(currentYellow==0):
                Signals[i].green-=1
                Signals[i].totalGreenTime+=1
            Else:
                Signals[i].yellow-=1
        Else:
            Signals[i].red-=1

# Generating vehicles in the simulation
Def generateVehicles():
    While(True):
        Vehicle_type = random.randint(0,4)
        If(vehicle_type==4):
            Lane_number = 0
        Else:
```

```
            Lane_number = random.randint(0,1) + 1
        Will_turn = 0
        If(lane_number==2):
            Temp = random.randint(0,4)
            If(temp<=2):
                Will_turn = 1
            Elif(temp>2):
                Will_turn = 0
        Temp = random.randint(0,999)
        Direction_number = 0
        A = [400,800,900,1000]
        If(temp<a[0]):
            Direction_number = 0
        Elif(temp<a[1]):
            Direction_number = 1
        Elif(temp<a[2]):
            Direction_number = 2
        Elif(temp<a[3]):
            Direction_number = 3
        Vehicle(lane_number, vehicleTypes[vehicle_type], direction_number,
directionNumbers[direction_number], will_turn)
        Time.sleep(0.75)


Def simulationTime():
    Global timeElapsed, simTime
    While(True):
        timeElapsed += 1
        time.sleep(1)
        if(timeElapsed==simTime):
            totalVehicles = 0
            print('Lane-wise Vehicle Counts')
            for I in range(noOfSignals):
                print('Lane',i+1,':',vehicles[directionNumbers[i]]['crossed'])
                totalVehicles += vehicles[directionNumbers[i]]['crossed']
```

```
        print('Total vehicles passed: ',totalVehicles)
        print('Total time passed: ',timeElapsed)
        print('No. of vehicles passed per unit time: ',(float(totalVehicles)/float(timeElapsed)))
        os._exit(1)


class Main:
    thread4 = threading.Thread(name="simulationTime",target=simulationTime, args=())
    thread4.daemon = True
    thread4.start()

    thread2 = threading.Thread(name="initialization",target=initialize, args=())    # initialization
    thread2.daemon = True
    thread2.start()

    # Colours
    Black = (0, 0, 0)
    White = (255, 255, 255)

    # Screensize
    screenWidth = 1400
    screenHeight = 800
    screenSize = (screenWidth, screenHeight)

    # Setting background image i.e. image of intersection
    Background = pygame.image.load('images/mod_int.png')

    Screen = pygame.display.set_mode(screenSize)
    Pygame.display.set_caption("SIMULATION")

    # Loading signal images and font
    redSignal = pygame.image.load('images/signals/red.png')
    yellowSignal = pygame.image.load('images/signals/yellow.png')
```

```
    greenSignal = pygame.image.load('images/signals/green.png')
    font = pygame.font.Font(None, 30)

    thread3 = threading.Thread(name="generateVehicles",target=generateVehicles, args=())    # Generating vehicles
    thread3.daemon = True
    thread3.start()

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()

        screen.blit(background,(0,0))   # display background in simulation
        for I in range(0,noOfSignals):  # display signal and set timer according to current status: green, yello, or red
            if(i==currentGreen):
                if(currentYellow==1):
                    if(signals[i].yellow==0):
                        signals[i].signalText = "STOP"
                    else:
                        signals[i].signalText = signals[i].yellow
                    screen.blit(yellowSignal, signalCoods[i])
                else:
                    if(signals[i].green==0):
                        signals[i].signalText = "SLOW"
                    else:
                        signals[i].signalText = signals[i].green
                    screen.blit(greenSignal, signalCoods[i])
            else:
                if(signals[i].red<=10):
                    if(signals[i].red==0):
                        signals[i].signalText = "GO"
                    else:
```

```
                signals[i].signalText = signals[i].red
            else:
                signals[i].signalText = "---"
            screen.blit(redSignal, signalCoods[i])
        signalTexts = ["","","","",""]

        # display signal timer and vehicle count
        For I in range(0,noOfSignals):
            signalTexts[i] = font.render(str(signals[i].signalText), True, white, black)
            screen.blit(signalTexts[i],signalTimerCoods[i])
            displayText = vehicles[directionNumbers[i]]['crossed']
            vehicleCountTexts[i] = font.render(str(displayText), True, black, white)
            screen.blit(vehicleCountTexts[i],vehicleCountCoods[i])

        timeElapsedText = font.render(("Time Elapsed: "+str(timeElapsed)), True, black, white)
        screen.blit(timeElapsedText,(1100,50))

        # display the vehicles
        For vehicle in simulation:
            Screen.blit(vehicle.currentImage, [vehicle.x, vehicle.y])
            # vehicle.render(screen)
            Vehicle.move()
        Pygame.display.update()

Main()
Import cv2
From darkflow.net.build import  TFNet
Import matplotlib.pyplot as plt
Import os

Options={
   'model':'./cfg/yolo.cfg',        #specifying the path of model
   'load':'./bin/yolov2.weights',   #weights
```

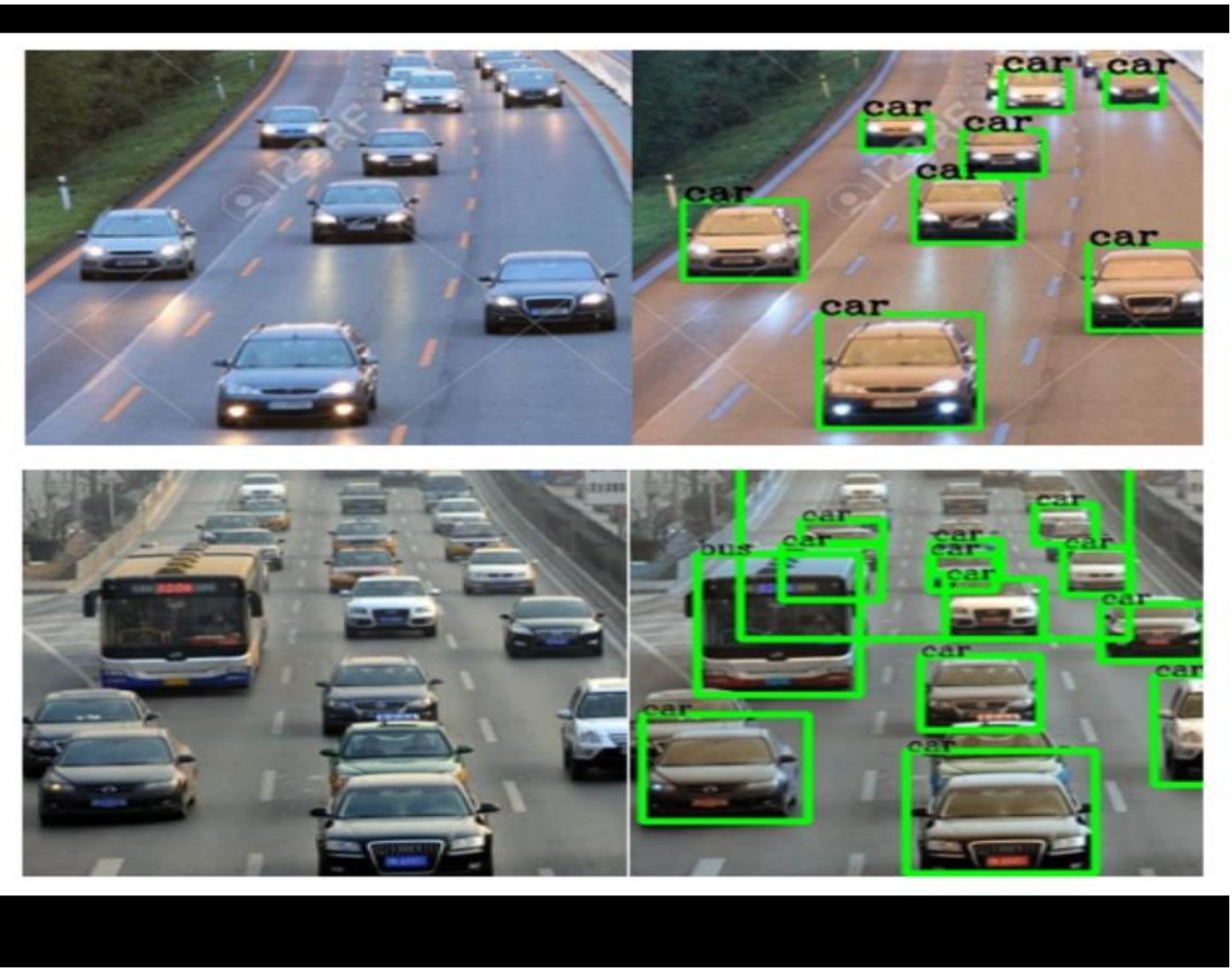
```
  'threshold':0.3                #minimum confidence factor to create a box, greater than 0.3 good
}

Tfnet=TFNet(options)
inputPath = os.getcwd() + "/test_images/"
outputPath = os.getcwd() + "/output_images/"

def detectVehicles(filename):
  global tfnet, inputPath, outputPath
  img=cv2.imread(inputPath+filename,cv2.IMREAD_COLOR)
  # img=cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
  Result=tfnet.return_predict(img)
  # print(result)
  For vehicle in result:
    Label=vehicle['label']   #extracting label
    If(label=="car" or label=="bus" or label=="bike" or label=="truck" or label=="rickshaw"):    #
drawing box and writing label
      Top_left=(vehicle['topleft']['x'],vehicle['topleft']['y'])
      Bottom_right=(vehicle['bottomright']['x'],vehicle['bottomright']['y'])
      Img=cv2.rectangle(img,top_left,bottom_right,(0,255,0),3)    #green box of width 5
      Img=cv2.putText(img,label,top_left,cv2.FONT_HERSHEY_COMPLEX,0.5,(0,0,0),1)
#image, label, position, font, font scale, colour: black, line width
  outputFilename = outputPath + "output_" +filename
  cv2.imwrite(outputFilename,img)
  print('Output image stored at:', outputFilename)
  # plt.imshow(img)
  # plt.show()
  # return result

For filename in os.listdir(inputPath):
  If(filename.endswith(".png") or filename.endswith(".jpg") or filename.endswith(".jpeg")):
    detectVehicles(filename)
print("Done!")
```

OUTPUT :


car
car
car
car
car
car
car
car
car
bus
car
car
car
car
car
car
car
car
car