TRAFFIC MANAGEMENT SYSTEM USING IOT

**PHASE 1: Problem Definition and Design Thinking**

****

**INTRODUCTION:**

* Traffic management system is one of the major proportions of a smart city.
* With the rapid growth of population and rapid increase of vehicles across the whole country which further leads to the traffic Congestion which is usually seen on roads.
* Nowadays traffic congestion is a difficult issue to deal with as number of vehicles is increasing day by day.
* To tackle various issues of traffic on roads and to help authorities in proper planning, a smart traffic management system using the Internet of Things (IoT).

**PROJECT DEFINITION:**

* The project involves using IoT devices and data analytics to monitor traffic flow and congestion in real-time, providing commuters with access to this information through a public platform or mobile apps.
* The objective is to help commuters make informed decisions about their routes and alleviate traffic congestion.
* This project includes defining objectives, designing the IoT traffic monitoring system, developing the traffic information platform, and integrating them using IoT technology and Python.

**DESIGN THINKING:**

1. **Project Objectives:**

Traffic control has as its principal objective to manage the movement of people and goods as efficiently and safely as possible. The dual objectives, however, frequently conflict or, at least, compete.

1. **IoT Sensor Design:**

Traffic Sensors (Doppler type)  
  
These sensors use the ultrasonic Doppler effect. They detect vehicles travelling in a particular direction using a change in frequency (the Doppler effect) according to the speed of the vehicle.

1. **Real-Time Transit Information Platform:**

Real Time Traffic Management systems manage traffic behaviours in real time by utilising a network of technologies including sensors, smart cameras, global positioning systems (GPS) and Bluetooth/Wi-Fi. This can be used to efficiently reduce congestion, bottlenecks and other traffic issues.

1. **Integration Approach:**

* The main modules in the proposed framework are as follows:
* A within-day module for TC;
* A choice model which implicitly simulates the RG effect in terms of compliance within the RG module;
* An iterative procedure for anticipatory route guidance which is able to model the consistency between user behavior and traffic signal decision variables, the consistency between information (*EsTTt*) and user behavior (*ETTt − 1*), and focus on information reliability (*ERt − 1*);
* A microscopic traffic flow model able to jointly simulate the effect of users’ preferences and decision variable optimizations

Objective:

* The primary objective of this project is to design and implement an IoT-based traffic management system that leverages advanced technologies to optimize traffic flow, reduce congestion, enhance safety, and improve overall transportation efficiency.
* By deploying a network of sensors and utilizing data analytics, this system aims to provide actionable insights to traffic operators and enable dynamic adjustments in traffic signal timings, ultimately leading to a smarter and more responsive traffic management.

Significance:

* Efficient Traffic Flow: Utilizing IoT, we aim to create an intelligent traffic management system that optimizes traffic flow by dynamically adapting traffic signals based on real-time data.
* Environmental Impact: By reducing traffic congestion and improving traffic efficiency, the system aims to decrease fuel consumption and greenhouse gas emissions, contributing to a greener environment.
* Safety Enhancement: Through real-time monitoring and incident detection, the system endeavors to enhance road safety by promptly responding to emergencies and minimizing potential accidents.
* Scalability and Adaptability: An IoT-based approach allows for scalable deployment and seamless integration of new sensors or technologies, ensuring the system can adapt to evolving urban landscapes and technological advancements.

System Architecture:

* Sensors and Data Acquisition Layer:
* *Traffic Flow Sensors:* Installed at strategic points to measure traffic density, vehicle speed, and direction.
* *Cameras:* Capture real-time images and videos for further analysis, including license plate recognition and incident detection.
* *Weather Sensors:* Gather data on weather conditions that may impact traffic.
* Data Processing and Edge Computing:
* *Microcontrollers and Edge Devices*: Receive data from sensors, process it locally, and filter relevant information before sending to the cloud.
* *Edge Analytics:* Perform initial data analysis and aggregation at the edge to reduce latency and bandwidth requirements.
* Communication Layer:
* *IoT Gateway:* Collects and aggregates data from edge devices and forwards it to the cloud for further processing.
* *Connectivity Protocols:* Utilize communication protocols (e.g., MQTT, HTTP, CoAP) to ensure seamless data transmission between edge devices, gateways, and the cloud.
* Cloud Infrastructure:
* *IoT Platform:* Manages device communication, data ingestion, storage, and provides APIs for data access.
* *Database:* Stores the collected and processed data for historical analysis and real-time decision-making.
* *Analytics Engine:* Utilizes machine learning algorithms and data analytics to process the data and extract actionable insights for traffic optimization.
* Traffic Management and Control:
* *Traffic Signal Control Algorithm:* An algorithm that processes the data from the analytics engine to dynamically adjust traffic signal timings for optimal traffic flow.
* *User Interface:* A web or mobile application that allows traffic operators to monitor traffic, view analytics, and manually control traffic signals if needed.
* Alerts and Notifications:
* *Notification Service:* Sends alerts to traffic operators or authorities in case of emergencies, accidents, or unusual traffic patterns.
* Security and Privacy:
* *Authentication and Authorization:* Ensure secure access to the system through proper authentication and authorization mechanisms.
* *Data Encryption:* Implement encryption protocols to secure data during transmission and storage.

Sensors and Data Collection:

* Traffic Flow Sensors:
* Inductive Loop Sensors: Embedded in roadways, detect changes in inductance caused by passing vehicles to measure traffic flow and vehicle speed.
* Infrared Sensors: Detect vehicle presence and count by emitting and sensing infrared light as vehicles pass by.
* Ultrasonic Sensors: Measure distance to detect vehicle presence and traffic density.
* Cameras:
* Video Cameras: Monitor traffic, record videos, and employ computer vision for vehicle detection, license plate recognition, and traffic analysis.
* Smart Cameras: Equipped with image processing capabilities to analyze traffic flow, detect congestion, and identify incidents.
* Weather Sensors:
* Anemometers: Measure wind speed and direction, critical for assessing how wind affects vehicle stability and traffic flow.
* Rain Sensors: Detect rainfall intensity, aiding in understanding its impact on traffic patterns and road conditions.
* Temperature and Humidity Sensors: Monitor environmental conditions affecting road safety and traffic behavior.
* Traffic Signal Sensors:
* Optical Sensors: Detect the presence of vehicles at traffic signals, assisting in optimizing signal timing based on traffic demand.
* GPS Devices:
* Vehicle GPS Trackers: Installed in vehicles to collect real-time location data, enabling traffic monitoring, route planning, and congestion analysis.
* Acoustic Sensors:
* Sound Sensors: Identify traffic noise patterns, honking, or accidents, contributing to traffic monitoring and incident detection.
* Pedestrian Sensors:
* Infrared or Ultrasonic Sensors:Detect pedestrian presence at crosswalks, enabling adjustments in traffic light timings for pedestrian safety.
* Mobile Apps and Crowdsourcing:
* Smartphone Sensors:Utilize smartphone sensors (GPS, accelerometer) through mobile apps to gather real-time traffic data from users, contributing to traffic monitoring and analysis.
* Radio Frequency Identification (RFID) Systems:
* RFID Readers: Identify and track vehicles using RFID tags, valuable for toll collection, vehicle identification, and traffic monitoring.
* License Plate Recognition (LPR) Systems:
* LPR Cameras:Capture images of license plates, enabling vehicle identification and monitoring.

IoT Platform in Traffic Management:

* Data Ingestion and Processing:
* Data Ingestion: Allows for efficient and secure ingestion of data from traffic sensors, cameras, and other devices.
* Data Pre-processing: Filters, cleans, and aggregates incoming data to prepare it for analysis and storage.
* Device Management:
* Device Registry: Maintains a registry of all connected devices, managing their metadata, configurations, and authentication.
* Security and Access Control: Implements security measures to ensure authorized access and secure communication between devices and the platform.
* Connectivity and Communication:
* Support for Protocols: Offers compatibility with various communication protocols (e.g., MQTT, CoAP , HTTP) to facilitate communication between devices and the platform.
* Bi-directional Communication: Allows devices to send data to the platform and receive commands or configurations in return.
* Data Storage and Management:
* Database: Stores collected data efficiently, allowing for retrieval, analysis, and historical data comparisons.
* Time-series Data Storage: Essential for storing and analyzing time-stamped traffic data, facilitating insights into traffic patterns and trends.
* Real-time Analytics:
* Analytics Engine: Provides real-time processing and analysis of traffic data, using algorithms to detect patterns, congestion, and anomalies.
* Rule Engine: Enables the definition of rules and triggers based on traffic conditions, allowing for immediate responses and alerts.
* Integration and APIs:
* API Endpoints: Offers APIs for easy integration with external applications, traffic management algorithms, and user interfaces.
* Integration with External Systems: Supports integration with third-party applications, traffic management software, and city infrastructure.
* Scalability and Flexibility:
* Scalable Infrastructure: Allows the platform to scale horizontally and vertically to handle growing amounts of traffic data and connected devices.
* Modular Architecture:Facilitates customization and expansion of functionalities to adapt to changing requirements.
* Dashboard and User Interface:
* Visualization Tools: Provides interactive dashboards and visualization tools for traffic operators to monitor traffic in real time.
* Alerts and Notifications: Sends alerts and notifications to traffic operators in case of traffic incidents or abnormal traffic patterns.

Algorithms for Traffic Pattern Analysis and Management:

1. K-means Clustering:

* Purpose: Group traffic data into clusters based on similarities (e.g., traffic density, vehicle speed) to identify traffic patterns in different areas.
* Application: Helps in understanding traffic patterns and congestion levels in different regions, aiding targeted traffic management strategies.

1. Density-based Spatial Clustering of Applications with Noise (DBSCAN):

* Purpose: Clusters traffic data points based on spatial density to identify congested areas and outliers.
* Application: Useful for detecting traffic congestion and abnormal traffic behavior, enabling timely interventions.

1. Kalman Filters:

* Purpose: Predicts future traffic states based on current and past traffic data, aiding in real-time traffic flow predictions.
* Application: Used for estimating traffic density, velocity, and acceleration to optimize traffic signal timings.

1. Shortest Path Algorithms (e.g., Dijkstra’s, A\*):

* Purpose: Determines the shortest or quickest path between two points, assisting in route optimization for traffic diversions and emergency response.
* Application: Helps in suggesting the optimal routes for vehicles to reduce travel time and avoid congested areas.

1. Traffic Signal Optimization Algorithms:

* Purpose: Dynamically adjust traffic signal timings based on real-time traffic data to optimize traffic flow and reduce congestion.
* Application: Enhances traffic flow at intersections by minimizing waiting times and improving overall traffic efficiency.

1. Neural Networks (e.g., LSTM, CNN):

* Purpose: Utilizes deep learning to model and predict traffic patterns, congestion, and traffic flow based on historical and real-time traffic data.
* Application: Helps in forecasting traffic conditions, enabling proactive traffic management strategies.

1. Queue Length Estimation Algorithms:

* Purpose: Estimate the length of vehicle queues at traffic signals or congested areas to optimize signal timings and minimize congestion.
* Application: Enhances traffic light control by adjusting timings based on the length of queues to improve traffic flow.

1. Genetic Algorithms:

* Purpose: Optimize traffic signal timings by simulating natural selection to find the most efficient timing plan for traffic lights.
* Application: Aids in improving traffic flow and minimizing waiting times at intersections.

1. Reinforcement Learning (e.g., Q-Learning):

* Purpose: Allows traffic control systems to learn and adapt traffic signal timings based on trial-and-error learning from traffic feedback.
* Application: Enables the traffic management system to continually improve traffic signal control policies over time.

User Interface in Traffic Management IOT Project:

* Dashboard:
* Real-time Traffic Overview: Display current traffic conditions, congestion levels, and weather updates in a clear and visually appealing manner.
* Map View: Present an interactive map showing traffic flow, incidents, and signal statuses at different intersections.
* Traffic Analytics:
* Traffic Patterns: Visualize traffic patterns and historical data for informed decision-making.
* Congestion Heat maps: Highlight congested areas using color-coded heatmaps for quick identification.
* Traffic Signal Control:
* Manual Override: Allow traffic operators to manually control traffic signals if needed, with options to adjust timings and priority settings.
* Automated Optimization: Display automated traffic signal optimization options based on real-time traffic data.
* Alerts and Notifications:
* Real-time Alerts: Notify operators of traffic incidents, accidents, or unusual traffic behavior requiring immediate attention.
* Customizable Alerts: Allow customization of alert preferences based on severity and location.
* Traffic Event Log:
* Incident Reporting: Enable operators to log incidents, accidents, and road closures with details for future reference and analysis.
* Historical Event Viewer: Provide a log of past events for review and analysis.
* Route Planning:
* Optimal Route Suggestions: Assist operators in planning optimal routes for emergency response or traffic diversion based on real-time traffic data.
* Traffic-Enabled Navigation: Enable operators to view and choose routes considering traffic conditions.
* Weather Information:
* Current Weather: Display real-time weather information that could impact traffic, such as rain, snow, or strong winds.
* Weather Forecasts: Provide short-term and long-term weather forecasts to aid in proactive traffic management.
* User Management:
* Authentication and Authorization: Implement secure login mechanisms and role-based access control for different users (e.g., admin, traffic operators).
* Performance Metrics:
* Key Performance Indicators (KPIs): Showcase metrics like average traffic speed, traffic density, and overall traffic system efficiency.
* Historical Comparison: Allow comparison of current KPIs with historical data for trend analysis.
* Search and Filtering:
* Search Functionality: Allow operators to search for specific locations, incidents.

Program:

Code:

Import numpy as np

Import pandas as pd

From sklearn.preprocessing import MinMaxScaler

From tensorflow.keras.models import Sequential

From tensorflow.keras.layers import LSTM, Dense

# Load the dataset

Df = pd.read\_csv(‘traffic\_data.csv’)

Data = df[‘vehicle\_count’].values.reshape(-1, 1)

# Normalize the data

Scaler = MinMaxScaler()

Data\_normalized = scaler.fit\_transform(data)

# Function to create sequences for LSTM

Def create\_sequences(data, seq\_length):

Sequences, labels = [], []

For I in range(len(data) – seq\_length):

Seq = data[i:i+seq\_length]

Label = data[i+seq\_length:i+seq\_length+1]

Sequences.append(seq)

Labels.append(label)

Return np.array(sequences), np.array(labels)

# Prepare the dataset with sequences

Sequence\_length = 5 # You can adjust this based on your requirement

X, y = create\_sequences(data\_normalized, sequence\_length)

# Split data into training and testing sets

Train\_size = int(len(X) \* 0.7)

X\_train, X\_test, y\_train, y\_test = X[:train\_size], X[train\_size:], y[:train\_size], y[train\_size:]

# Build the LSTM model

Model = Sequential()

Model.add(LSTM(units=100, activation=’relu’, input\_shape=(sequence\_length, 1)))

Model.add(Dense(units=1))

Model.compile(optimizer=’adam’, loss=’mean\_squared\_error’)

# Train the model

Model.fit(X\_train, y\_train, epochs=50, batch\_size=32)

# Predict using the model

Predictions = model.predict(X\_test)

Predictions = scaler.inverse\_transform(predictions)

# Monitor traffic predictions

For i in range(len(predictions)):

Print(f”Predicted vehicle count at time {I + train\_size + sequence\_length}: {predictions[i][0]:.2f}”)

CODING PHASE

*PROGRAM:*

import cv2

import dlib

import time

import threading

import math

import helm

carCascade = cv2.CascadeClassifier('cars.xml')

bikeCascade = cv2.CascadeClassifier('motor-v4.xml')

video = cv2.VideoCapture('test.mp4')

LAG=7

WIDTH = 1280

HEIGHT = 720

OPTIMISE= 7

def estimateSpeed(location1, location2,fps):

d\_pixels = math.sqrt(math.pow(location2[0] - location1[0], 2) + math.pow(location2[1] - location1[1], 2))

# ppm = location2[2] / carWidht

ppm = 8.8

d\_meters = d\_pixels / ppm

if fps == 0.0:

fps = 18

speed = d\_meters \* fps \* 3.6

return speed

def trackMultipleObjects():

rectangleColor = (0, 255, 0)

frameCounter = 0

currentCarID = 0

currentBikeID=0

fps = 0

carTracker = {}

bikeTracker = {}

bikeNumbers = {}

carNumbers = {}

bikeLocation1 = {}

carLocation1 = {}

bikeLocation2 = {}

carLocation2 = {}

speed = [None] \* 1000

go =[False for i in range(1000)]

identity = [0 for i in range(1000)]

snaps = [False for i in range(1000)]

types = ["cars" for i in range(1000)]

Helmets = ["No Helmet Detected" for i in range(1000)]

out = cv2.VideoWriter('outpy.avi',cv2.VideoWriter\_fourcc('M','J','P','G'), 10, (WIDTH,HEIGHT))

while True:

start\_time = time.time()

rc, image = video.read()

if type(image) == type(None):

break

image = cv2.resize(image, (WIDTH, HEIGHT))

resultImage = image.copy()

frameCounter = frameCounter + 1

carIDtoDelete = []

for carID in carTracker.keys():

trackingQuality = carTracker[carID].update(image)

if trackingQuality < 7:

carIDtoDelete.append(carID)

for carID in carIDtoDelete:

print ('Removing carID ' + str(carID) + ' from list of trackers.')

print ('Removing carID ' + str(carID) + ' previous location.')

print ('Removing carID ' + str(carID) + ' current location.')

carTracker.pop(carID, None)

carLocation1.pop(carID, None)

carLocation2.pop(carID, None)

if not (frameCounter % 10):

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

cars = carCascade.detectMultiScale(gray, 1.1, 13, 18, (24, 24))

bikes = bikeCascade.detectMultiScale(gray, 1.1 , 13, 18, (24,24))

for (\_x, \_y, \_w, \_h) in cars:

x = int(\_x)

y = int(\_y)

w = int(\_w)

h = int(\_h)

roi = image[y:y+h,x:x+w]

x\_bar = x + 0.5 \* w

y\_bar = y + 0.5 \* h

matchCarID = None

for carID in carTracker.keys():

trackedPosition = carTracker[carID].get\_position()

t\_x = int(trackedPosition.left())

t\_y = int(trackedPosition.top())

t\_w = int(trackedPosition.width())

t\_h = int(trackedPosition.height())

t\_x\_bar = t\_x + 0.5 \* t\_w

t\_y\_bar = t\_y + 0.5 \* t\_h

if ((t\_x <= x\_bar <= (t\_x + t\_w)) and (t\_y <= y\_bar <= (t\_y + t\_h)) and (x <= t\_x\_bar <= (x + w)) and (y <= t\_y\_bar <= (y + h))):

matchCarID = carID

if matchCarID is None:

print ('Creating new tracker ' + str(currentCarID))

tracker = dlib.correlation\_tracker()

tracker.start\_track(image, dlib.rectangle(x, y, x + w, y + h))

carTracker[currentCarID] = tracker

carLocation1[currentCarID] = [x, y, w, h]

currentCarID = currentCarID + 1

for (\_x, \_y, \_w, \_h) in bikes:

x = int(\_x)

y = int(\_y)

w = int(\_w)

h = int(\_h)

x\_bar = x + 0.5 \* w

y\_bar = y + 0.5 \* h

matchCarID = None

for carID in carTracker.keys():

trackedPosition = carTracker[carID].get\_position()

t\_x = int(trackedPosition.left())

t\_y = int(trackedPosition.top())

t\_w = int(trackedPosition.width())

t\_h = int(trackedPosition.height())

t\_x\_bar = t\_x + 0.5 \* t\_w

t\_y\_bar = t\_y + 0.5 \* t\_h

if ((t\_x <= x\_bar <= (t\_x + t\_w)) and (t\_y <= y\_bar <= (t\_y + t\_h)) and (x <= t\_x\_bar <= (x + w)) and (y <= t\_y\_bar <= (y + h))):

matchCarID = carID

if matchCarID is None:

print ('Creating new tracker ' + str(currentCarID))

tracker = dlib.correlation\_tracker()

tracker.start\_track(image, dlib.rectangle(x, y, x + w, y + h))

carTracker[currentCarID] = tracker

carLocation1[currentCarID] = [x, y, w, h]

types[currentCarID]= "bikes"

currentCarID = currentCarID + 1

for carID in carTracker.keys():

trackedPosition = carTracker[carID].get\_position()

t\_x = int(trackedPosition.left())

t\_y = int(trackedPosition.top())

t\_w = int(trackedPosition.width())

t\_h = int(trackedPosition.height())

cv2.rectangle(resultImage, (t\_x, t\_y), (t\_x + t\_w, t\_y + t\_h), rectangleColor, 4)

carLocation2[carID] = [t\_x, t\_y, t\_w, t\_h]

end\_time = time.time()

fps=0.0

for i in carLocation1.keys():

if frameCounter % 1 == 0:

[x1, y1, w1, h1] = carLocation1[i]

[x2, y2, w2, h2] = carLocation2[i]

carLocation1[i] = [x2, y2, w2, h2]

if [x1, y1, w1, h1] != [x2, y2, w2, h2]:

result = False

roi = resultImage[y1:y1+h1,x1:x1+w1]

if types[i]=="bikes" and Helmets[i] == "No Helmet Detected" and identity[i]< OPTIMISE:

result = helm.detect(roi)

if result==True:

Helmets[i]= "Helmet Detected"

if 7==7:

if not (end\_time == start\_time):

fps = 1.0/(end\_time - start\_time)

speed[i] = estimateSpeed([x1, y1, w1, h1], [x2, y2, w2, h2],fps)

if int(speed[i])>40:

speed[i]= speed[i]%40

if go[i] == True and int(speed[i])<10:

speed[i]=speed[i]+15

if int(speed[i])==0:

continue

if int(speed[i])>30:

go[i]=True

cv2.putText(resultImage, "OverSpeeding ALERT", (int(x1 + w1/2), int(y1-5)),cv2.FONT\_HERSHEY\_SIMPLEX, 0.75, (0, 0, 255), 2)

elif speed[i] != None and y1 >= 180 and speed[i]!=0:

ans= str(int(speed[i])) + " km/hr "

if types[i]=="bikes":

ans= ans+ Helmets[i]

cv2.putText(resultImage, ans, (int(x1 + w1/2), int(y1-5)),cv2.FONT\_HERSHEY\_SIMPLEX, 0.75, (0, 255, 0), 2)

identity[i]+=1

cv2.imshow('result', resultImage)

if cv2.waitKey(33) == 27:

break

cv2.destroyAllWindows()

if \_name\_ == '\_main\_':

trackMultipleObjects()

The main file to run the project…

from time import sleep

import cv2 as cv

import argparse

import sys

import numpy as np

import os.path

from glob import glob

#from PIL import image

frame\_count = 0 # used in mainloop where we're extracting images., and then to drawPred( called by post process)

frame\_count\_out=0 # used in post process loop, to get the no of specified class value.

# Initialize the parameters

confThreshold = 0.5 #Confidence threshold

nmsThreshold = 0.4 #Non-maximum suppression threshold

inpWidth = 416 #Width of network's input image

inpHeight = 416 #Height of network's input image

# Load names of classes

classesFile = "obj.names";

classes = None

with open(classesFile, 'rt') as f:

classes = f.read().rstrip('\n').split('\n')

# Give the configuration and weight files for the model and load the network using them.

modelConfiguration = "yolov3-obj.cfg";

modelWeights = "yolov3-obj\_2400.weights";

net = cv.dnn.readNetFromDarknet(modelConfiguration, modelWeights)

net.setPreferableBackend(cv.dnn.DNN\_BACKEND\_OPENCV)

net.setPreferableTarget(cv.dnn.DNN\_TARGET\_CPU)

# Get the names of the output layers

def getOutputsNames(net):

# Get the names of all the layers in the network

layersNames = net.getLayerNames()

# Get the names of the output layers, i.e. the layers with unconnected outputs

return [layersNames[i-1] for i in net.getUnconnectedOutLayers()]

# Draw the predicted bounding box

def drawPred(classId, conf, left, top, right, bottom, frame):

global frame\_count

# Draw a bounding box.

#cv.rectangle(frame, (left, top), (right, bottom), (255, 178, 50), 3)

label = '%.2f' % conf

# Get the label for the class name and its confidence

if classes:

assert(classId < len(classes))

label = '%s:%s' % (classes[classId], label)

#Display the label at the top of the bounding box

labelSize, baseLine = cv.getTextSize(label, cv.FONT\_HERSHEY\_SIMPLEX, 0.5, 1)

top = max(top, labelSize[1])

#print(label) #testing

#print(labelSize) #testing

#print(baseLine) #testing

label\_name,label\_conf = label.split(':') #spliting into class & confidance. will compare it with person.

if label\_name == 'Helmet':

#will try to print of label have people.. or can put a counter to find the no of people occurance.

#will try if it satisfy the condition otherwise, we won't print the boxes or leave it.

#cv.rectangle(frame, (left, top - round(1.5\*labelSize[1])), (left + round(1.5\*labelSize[0]), top + baseLine), (255, 255, 255), cv.FILLED)

#cv.putText(frame, label, (left, top), cv.FONT\_HERSHEY\_SIMPLEX, 0.75, (0,0,0), 1)

frame\_count+=1

#print(frame\_count)

if(frame\_count> 0):

return frame\_count

# Remove the bounding boxes with low confidence using non-maxima suppression

def postprocess(frame, outs):

frameHeight = frame.shape[0]

frameWidth = frame.shape[1]

frame\_count\_out=0

classIds = []

confidences = []

boxes = []

# Scan through all the bounding boxes output from the network and keep only the

# ones with high confidence scores. Assign the box's class label as the class with the highest score.

classIds = [] #have to fins which class have hieghest confidence........=====>>><<<<=======

confidences = []

boxes = []

for out in outs:

for detection in out:

scores = detection[5:]

classId = np.argmax(scores)

confidence = scores[classId]

if confidence > confThreshold:

center\_x = int(detection[0] \* frameWidth)

center\_y = int(detection[1] \* frameHeight)

width = int(detection[2] \* frameWidth)

height = int(detection[3] \* frameHeight)

left = int(center\_x - width / 2)

top = int(center\_y - height / 2)

classIds.append(classId)

#print(classIds)

confidences.append(float(confidence))

boxes.append([left, top, width, height])

# Perform non maximum suppression to eliminate redundant overlapping boxes with

# lower confidences.

indices = cv.dnn.NMSBoxes(boxes, confidences, confThreshold, nmsThreshold)

count\_person=0 # for counting the classes in this loop.

for i in indices:

i = i[0]

box = boxes[i]

left = box[0]

top = box[1]

width = box[2]

height = box[3]

#this function in loop is calling drawPred so, try pushing one test counter in parameter , so it can calculate it.

frame\_count\_out = drawPred(classIds[i], confidences[i], left, top, left + width, top + height, frame)

#increase test counter till the loop end then print...

#checking class, if it is a person or not

my\_class='Helmet'

mycode .....

unknown\_class = classes[classId]

if my\_class == unknown\_class:

count\_person += 1

#if(frame\_count\_out > 0):

#print(frame\_count\_out)

if count\_person >= 1:

path = 'test\_out/'

# frame\_name=os.path.basename(fn) # trimm the path and give file name.

#cv.imwrite(str(path)+frame\_name, frame) # writing to folder.

#print(type(frame))

#cv.imshow('img',frame)

#cv.waitKey(800)

return 1

else:

return 0

#cv.imwrite(frame\_name, frame)

# Process inputs

winName = 'Deep learning object detection in OpenCV'

cv.namedWindow(winName, cv.WINDOW\_NORMAL)

def detect(frame):

#frame = cv.imread(fn)

frame\_count =0

# Create a 4D blob from a frame.

blob = cv.dnn.blobFromImage(frame, 1/255, (inpWidth, inpHeight), [0,0,0], 1, crop=False)

# Sets the input to the network

net.setInput(blob)

# Runs the forward pass to get output of the output layers

outs = net.forward(getOutputsNames(net))

# Remove the bounding boxes with low confidence

# Put efficiency information. The function getPerfProfile returns the overall time for inference(t) and the timings for each of the layers(in layersTimes)

t, \_ = net.getPerfProfile()

#print(t)

label = 'Inference time: %.2f ms' % (t \* 1000.0 / cv.getTickFrequency())

#print(label)

#cv.putText(frame, label, (0, 15), cv.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 0, 255))

#print(label)

k=postprocess(frame, outs)

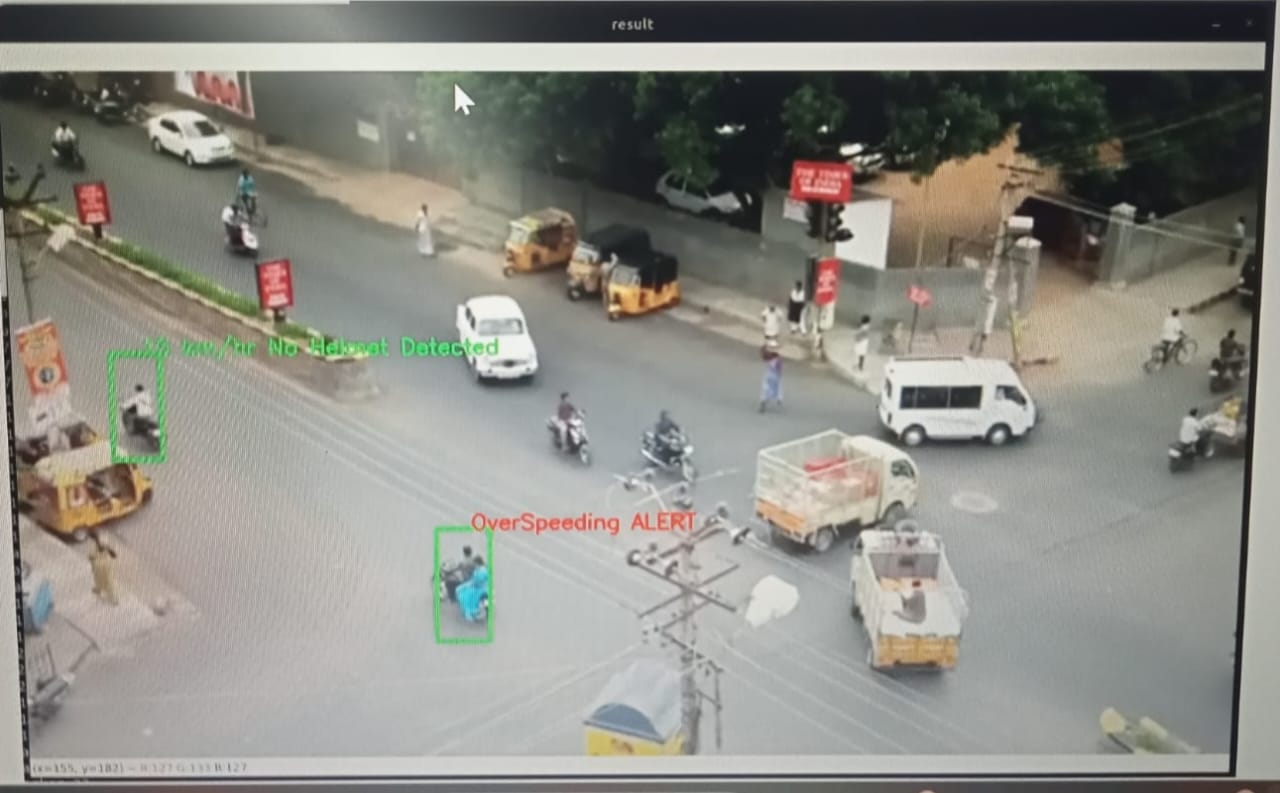
if k:

return 1

else:

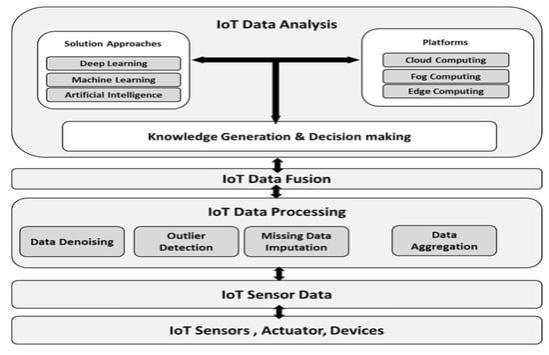
return 0

***OUTPUT:***



COMMUNICATION BETWEEN IOT HARDWARE AND THE SOFTWARE :

* IoT can also use artificial intelligence and machine learning to aid in making data collection processes easier and more dynamic. An IoT system collects data from sensors installed in IoT devices and transfers that data through an IoT gateway for it to be analyzed by an application or back-end system.



* On the Internet of Things (IoT), data from multiple sensors is typically managed using a combination of hardware, software, and networking solutions.
* First, the data from the sensors is collected and processed by hardware devices, such as gateway devices or edge computers. These devices are typically located at the point where the sensors are deployed and are responsible for collecting and aggregating the data from the sensors. The hardware devices may also perform some initial processing and filtering of the data to remove any irrelevant or incorrect information.
* Next, the data from the hardware devices is transmitted to a central server or cloud platform, where it is stored and managed. This typically involves using a network, such as a cellular or Wi-Fi network, to transmit the data from the hardware devices to the central server. The data is then stored in a database or data lake, where it can be accessed and analyzed by various applications and tools.
* Finally, the data from the sensors is typically accessed and analyzed using a combination of software tools and applications. These tools and applications may include dashboards, analytics software, and machine learning algorithms, which can help to visualize, analyze, and make sense of the data.
* There are many different software tools that can be used to analyze data from sensors in the Internet of Things (IoT). Some examples of these tools include:
* Data visualization tools, such as dashboards and charts, which can help to visualize and understand the data in a graphical format.
* Analytics software, such as business intelligence and predictive analytics tools, which can help to analyze the data and identify trends, patterns, and anomalies.
* Machine learning algorithms, which can be used to build predictive models and make predictions based on the data.
* Stream processing and real-time analytics tools, which can help to process and analyze high volumes of data in real time.
* Data management and storage solutions, such as databases and data lakes, which can help to store and manage the data from the sensors.
* The specific tools that are used will depend on the type and volume of data, as well as the specific goals and objectives of the analysis.
* Overall, managing data from multiple sensors in the IoT involves a combination of hardware, software, and networking solutions, which work together to collect, transmit, store, and analyze the data from the sensors.

OVERVIEW :

The following is an overview of the process of building a house price prediction model by feature selection, model training, and evaluation:

1. **Problem Definition**:
   * Clearly define the problem you want to solve with the traffic management system. Identify specific objectives, such as reducing congestion, improving safety, or optimizing traffic flow.
2. **Data Collection**:
   * Collect real-time traffic data using IoT sensors and devices. This data can include information about vehicle counts, speeds, weather conditions, and road infrastructure.
3. **Feature Selection**:
   * Determine which features (variables) are most relevant for your traffic management model. Feature selection is crucial for reducing data dimensionality and improving model performance.
   * Consider using techniques like correlation analysis, feature importance scores, or domain knowledge to choose the most important features.
4. **Data Preprocessing**:
   * Clean and preprocess the data to handle missing values, outliers, and data quality issues. This may involve data imputation, normalization, and scaling.
5. **Model Selection**:
   * Choose an appropriate machine learning or deep learning model for your traffic management task. Some common models for this type of application include decision trees, random forests, neural networks, or recurrent neural networks (RNNs).
6. **Model Training**:
   * Split your data into training and testing sets to train and validate your model. The training process involves feeding the data to the model and adjusting its parameters to optimize performance.
7. **Model Evaluation**:
   * Assess the performance of your model using appropriate evaluation metrics. Common metrics for traffic management systems include Mean Absolute Error (MAE), Mean Squared Error (MSE), or custom-defined metrics based on your specific objectives.
   * Consider using techniques like cross-validation to ensure your model's robustness.
8. **Deployment**:
   * Once your model performs well, deploy it in a real-world traffic management system. Ensure that IoT devices collect data continuously and send it to the model for real-time predictions.
9. **Feedback Loop**:
   * Establish a feedback mechanism to continuously monitor and update the model's performance. As new data becomes available, retrain the model to adapt to changing traffic conditions.
10. **Integration with Control Systems**:
    * Connect the traffic management model with control systems such as traffic signals, dynamic message signs, or autonomous vehicles to implement traffic control strategies based on the model's predictions.
11. **Optimization and Tuning**:
    * Continuously optimize and fine-tune the system based on the real-world outcomes and feedback.
12. **Monitoring and Maintenance**:
    * Regularly monitor the IoT devices, model performance, and overall system functionality. Maintenance and updates are essential for long-term success.
13. **Compliance and Privacy**:
    * Ensure that your traffic management system complies with relevant regulations and addresses privacy concerns, especially when collecting and processing sensitive data.
14. **Scalability**:
    * Plan for scalability as traffic volumes and system complexity may increase over time. Ensure that your infrastructure can handle larger datasets and more IoT devices.
15. **User Interface**:
    * Develop user-friendly interfaces for traffic operators and stakeholders to interact with the system and access valuable insights

PROCEDURE:

FEATURE SELECTION:

Feature selection for traffic management using IoT involves choosing the most relevant data points or attributes to optimize the performance of an IoT-based traffic management system. Here's how you can approach it:

1. **Understand the Problem:** Start by defining the specific goals of your traffic management system. What are you trying to optimize or predict? Common objectives include reducing congestion, improving traffic flow, and enhancing safety.
2. **Data Collection:** Gather data from IoT devices, such as traffic cameras, sensors, GPS trackers, and weather stations. This data can include information on vehicle speed, traffic volume, weather conditions, road conditions, and more.
3. **Data Preprocessing:** Before selecting features, preprocess the data to handle missing values, outliers, and noise. Normalize or scale the data if necessary.
4. **Feature Selection Techniques:** There are various feature selection techniques you can use:

a. **Filter Methods:** These methods rank features based on statistical metrics like correlation, chi-squared, or mutual information. You can choose the top-ranked features.

b. **Wrapper Methods:** These methods involve training a model with different subsets of features and evaluating their performance. Common techniques include recursive feature elimination (RFE) and forward selection.

c. **Embedded Methods:** Some machine learning algorithms have built-in feature selection, such as L1 regularization in linear regression or tree-based models like Random Forests.

1. **Domain Knowledge:** Consider domain expertise and consult with traffic engineers or experts to identify critical features. They can provide insights into which data is most relevant for traffic management.
2. **Dimensionality Reduction:** In some cases, you may use techniques like Principal Component Analysis (PCA) to reduce the dimensionality of the data while retaining important information.
3. **Regular Updates:** Keep in mind that traffic patterns can change over time, so regularly reevaluate your feature selection to adapt to evolving conditions.
4. **Model Building:** Once you've selected the most relevant features, build your traffic management model, which could be based on machine learning algorithms or other techniques.
5. **Evaluation:** Evaluate the performance of your traffic management system using metrics relevant to your objectives, such as traffic flow efficiency, accident prediction accuracy, or congestion reduction.
6. **Iterate:** Continuously monitor and refine your feature selection and model as traffic conditions change and new data becomes available.

CODE:

import pandas as pd

from sklearn.feature\_selection import SelectKBest

from sklearn.feature\_selection import f\_classif

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

# Load your IoT sensor data into a DataFrame

data = pd.read\_csv('traffic\_sensor\_data.csv') # Replace 'traffic\_sensor\_data.csv' with your dataset file

# Assuming your dataset has features and a target variable (e.g., traffic congestion status)

X = data.drop('Traffic\_Status', axis=1) # Features

y = data['Traffic\_Status'] # Target variable

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Feature selection using SelectKBest with ANOVA F-value

k\_best = SelectKBest(score\_func=f\_classif, k=3) # Select the top 3 features

X\_train\_new = k\_best.fit\_transform(X\_train, y\_train)

X\_test\_new = k\_best.transform(X\_test)

# Train a classifier (e.g., Random Forest) on the selected features

clf = RandomForestClassifier()

clf.fit(X\_train\_new, y\_train)

# Evaluate the model

accuracy = clf.score(X\_test\_new, y\_test)

print("Accuracy on test data:", accuracy)

MODEL TRAINING:

CODE:

1.K-means Clustering:

import pandas as pd

from sklearn.cluster import KMeans

import matplotlib.pyplot as plt

# Load your preprocessed IoT traffic data (replace 'data.csv' with your dataset)

data = pd.read\_csv('data.csv')

# Assuming your dataset includes features like speed, density, and location

X = data[['speed', 'density', 'latitude', 'longitude']]

# Choose the number of clusters (k) based on your needs

k = 5

# Initialize and fit the K-Means model

kmeans = KMeans(n\_clusters=k, random\_state=0)

kmeans.fit(X)

# Add cluster labels to the original data

data['cluster\_label'] = kmeans.labels\_

# Visualize the clusters if needed

plt.scatter(X['latitude'], X['longitude'], c=kmeans.labels\_, cmap='rainbow')

plt.show()

# You can use the cluster labels for traffic management decisions

for cluster in range(k):

cluster\_data = data[data['cluster\_label'] == cluster]

# Implement traffic management strategies for each cluster

# Save or export the cluster labels and decisions for real-time use

data.to\_csv('clustered\_traffic\_data.csv', index=False)

2. A\* ALGORITHM:

import heapq

# Define a class for nodes in the search graph

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, cost=0, heuristic=0):

self.state = state # Current state (location)

self.parent = parent # Parent node

self.action = action # Action to reach this state from the parent

self.cost = cost # Cost to reach this state from the start

self.heuristic = heuristic # Heuristic estimate of remaining cost

def total\_cost(self):

return self.cost + self.heuristic

# A\* search function

def astar\_search(start, goal, heuristic\_fn):

open\_set = []

closed\_set = set()

start\_node = Node(state=start, cost=0, heuristic=heuristic\_fn(start))

heapq.heappush(open\_set, (start\_node.total\_cost(), start\_node))

while open\_set:

\_, current\_node = heapq.heappop(open\_set)

if current\_node.state == goal:

path = []

while current\_node:

path.insert(0, current\_node.state)

current\_node = current\_node.parent

return path

if current\_node.state in closed\_set:

continue

closed\_set.add(current\_node.state)

# Expand the current node and consider its neighbors

for neighbor, cost in get\_neighbors(current\_node.state):

if neighbor in closed\_set:

continue

new\_cost = current\_node.cost + cost

neighbor\_node = Node(state=neighbor, parent=current\_node, cost=new\_cost, heuristic=heuristic\_fn(neighbor))

heapq.heappush(open\_set, (neighbor\_node.total\_cost(), neighbor\_node))

return None # If no path is found

# Define a function to return neighbors and their costs (simplified for illustration)

def get\_neighbors(node):

# In a real scenario, you would retrieve data about road connections and costs

# Here, we assume a simple grid for illustration purposes

x, y = node

neighbors = [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)]

valid\_neighbors = [(x, y) for x, y in neighbors if 0 <= x < 5 and 0 <= y < 5]

return [(neighbor, 1) for neighbor in valid\_neighbors]

# Example usage

start = (0, 0)

goal = (4, 4)

# Define a simple heuristic function (Euclidean distance)

def heuristic(state):

return ((state[0] - goal[0]) \*\* 2 + (state[1] - goal[1]) \*\* 2) \*\* 0.5

path = astar\_search(start, goal, heuristic)

print(path)

3. CNN:

import tensorflow as tf

from tensorflow import keras

model = keras.Sequential([

keras.layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(image\_width, image\_height, 3)),

keras.layers.MaxPooling2D((2, 2)),

keras.layers.Flatten(),

keras.layers.Dense(128, activation='relu'),

keras.layers.Dense(num\_classes, activation='softmax')

])

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

MODEL TRAINING:

Training a model for traffic management using an IoT project typically involves a combination of data collection, data preprocessing, model selection, and training. Here's an outline of the model training process:

1. Data Collection:

* Collect traffic data using IoT sensors such as cameras, traffic counters, or environmental sensors.
* Gather data on vehicle counts, speeds, types, weather conditions, and any other relevant parameters.
* Ensure data quality, including cleaning and handling missing values.

2. Data Preprocessing:

* Normalize and scale the data to make it suitable for training.
* Perform feature engineering to create relevant features from raw sensor data.
* Split the dataset into training, validation, and test sets.

3. Data Labeling:

* Annotate or label the data. For example, label data points as "traffic congestion" or "normal traffic."
* You may need to manually label data or use unsupervised learning techniques to categorize traffic patterns.

4. Model Selection:

* Choose an appropriate machine learning or deep learning model based on your problem. Options may include decision trees, random forests, SVMs, or deep neural networks.
* Consider whether a supervised or unsupervised approach is suitable for your specific task.

5. Model Training:

* + Train your selected model on the labeled dataset using appropriate machine learning libraries such as scikit-learn or TensorFlow/Keras.
  + Tune hyperparameters to optimize model performance using techniques like grid search or random search.
  + Monitor training metrics (e.g., accuracy, F1 score, mean squared error) to assess model performance.

6. Evaluation:

* + Evaluate the trained model's performance on the validation dataset to assess its generalization capabilities.
  + Use metrics such as accuracy, precision, recall, or mean absolute error depending on the specific problem.

7. Model Fine-Tuning:

* + Adjust the model based on the validation results, retrain it if necessary, and iterate through the process to improve performance.

8. Testing:

* + Test the model on the separate test dataset to assess its performance on unseen data.
  + Ensure the model is not overfitting the training data.

9. Deployment:

* + Once the model performs well, deploy it within your IoT traffic management system.
  + Implement the model in a real-time or near-real-time environment to make traffic management decisions.

10. Continuous Monitoring and Maintenance:

* + Continuously monitor the model's performance in the deployed environment.
  + Re-train the model periodically with new data to adapt to changing traffic conditions and maintain accuracy.

11. Feedback Loop:

* + Use real-world feedback, such as traffic camera data, to continually improve the model's predictions and traffic management strategies.

MODEL EVALUATION:

Evaluating a traffic management system that uses IoT data and machine learning models is essential to assess its performance and make improvements. Here are some key aspects to consider when evaluating such a system:

1. Data Quality Evaluation:

- Start by assessing the quality of the data collected from IoT sensors. Check for issues such as missing data, sensor errors, and data inconsistencies.

- Implement data validation and cleaning procedures to ensure the reliability of your data.

2. Model Performance Evaluation:

- Evaluate the performance of the machine learning models used for traffic management. The specific metrics you use will depend on the nature of your model (e.g., classification or regression). Common metrics include accuracy, precision, recall, F1 score, mean squared error, and others.

3. Real-time Performance:

- Assess the system's real-time performance by measuring response times for data collection, model inference, and decision-making. Ensure that the system can make timely decisions in response to changing traffic conditions.

4. Congestion Management:

- Evaluate the system's effectiveness in managing traffic congestion. Compare the actual traffic conditions to the system's predictions and decisions. Measure the reduction in congestion and traffic delays.

5. Safety Assessment:

- Assess the safety of the traffic management decisions made by the system. Ensure that the system doesn't create unsafe conditions, such as sudden changes in traffic signal timings that could lead to accidents.

6. Resource Utilization:

- Evaluate the efficient use of resources, such as energy consumption for traffic signals, by the traffic management system. Optimize resource allocation for sustainability.

7. Scalability:

- Test the system's scalability to handle increasing data volume and complexity. Ensure that it can adapt to the needs of a growing city or region.

8. User Satisfaction:

- Collect feedback from users, such as commuters, city officials, and traffic engineers, to gauge their satisfaction with the system's performance. User feedback can be invaluable for making improvements.

9. False Positives and False Negatives:

- Analyze the number of false positives (incorrectly identifying congestion) and false negatives (failing to detect congestion). Balance these metrics to avoid unnecessary interventions and missed opportunities for traffic management.

10. Robustness:

- Test the system's robustness against adverse conditions, such as inclement weather, sensor failures, or sudden events (e.g., accidents or road closures).

11. Adaptability:

- Assess the system's ability to adapt to changing traffic patterns and conditions. Use historical data to validate its adaptability.

12. Cost-Benefit Analysis:

* + Conduct a cost-benefit analysis to determine the economic impact of the traffic management system. Calculate the savings in terms of reduced fuel consumption, travel time, and environmental benefits.

13. Legal and Regulatory Compliance:

* + Ensure that the traffic management system complies with local traffic regulations and data privacy laws.

14. Continuous Monitoring:

* + Implement continuous monitoring of the system's performance, as traffic conditions change over time. Update and fine-tune the system as needed.

15. Comparative Analysis:

* + Compare the performance of your IoT-based traffic management system with traditional traffic management methods to assess its advantages and disadvantages.

CONCLUSION:

* This system configuration reduces huge traffic queues caused by the conventionally implemented system used in many places.
* The system also additionally reduces the workload of officers who would have to direct traffic in unexpected situations, or when the traffic lights are not responding.
* It also enables traffic lights to work continuously with less chances of malfunctioning.
* The system in simple words provides a simple yet effective solution to improper traffic management systems.