KolKGP Convergence

I. Introduction to Problem Statement

A. Our Understanding

The problem statement aims to develop a predictive model for estimating vehicle counts by class in real-time traffic video clips. The model was to focus on two tasks: predicting the cumulative number of vehicles from the start to the end of a 30-minute clip, and forecasting vehicle counts for the subsequent 30 minutes. Additionally, the model would be tested on unseen locations, predicting vehicle counts during a new 30-minute clip and forecasting the cumulative number for the following 30 minutes. The model's performance in these tasks would provide insights into its ability to generalize across different traffic scenarios.

1) Problem 1:

- The cumulative observed number of vehicles by vehicle class from the start of the clip to the end of the clip in a given 30-minute clip (split into two 15-minute clips) for a given camera.
- The predicted cumulative number of vehicles by vehicle class from the end time of the given clip up to 30 minutes into the future.

2) Problem 2:

- The predicted number of vehicles by vehicle class at a previously unseen location from the starting time to the ending time of the 30-minute clip provided.
- The predicted cumulative number of vehicles by vehicle class at a previously unseen location from the end time of the 30-minute clips provided up to 30 minutes into the future.

II. Methodology

A. Model Architecture

YOLOv8 achieves state-of-the-art accuracy on various object detection benchmarks, boasts impressive inference speeds suitable for real-time applications such as autonomous vehicles and robotics, is lightweight and efficient requiring fewer computational resources making it ideal for deployment on edge devices, and is open-source with strong community support that fosters continuous development and improvement. Because of these attributes, YOLOv8 has been widely used for vehicle counting, given its effectiveness in real-time scenarios. The YOLOv8 Medium model, in particular, was chosen to balance performance and accuracy, making it an excellent fit for applications that demand both of these.

Fig. 1: Methodology Flowchart

III. Approach

This section deals with the detailed approach towards the problem statement.

A. Data Processing

Selected videos were chosen from the huge dataset provided and these videos were split into frames and around 1 frame every two seconds was saved to be annotated. Camera junctions were strategically chosen at different times to ensure variety of data across all cameras and times. The first task was to create a basic YOLOv8 model for seven vehicle classes (Car,Bus,Two-wheeler,Three-wheeler, LCV and Truck). Given that annotation is a laborious and time consuming process, Auto Annotation was used. An initial model was created which was further used to annotate more images. Then it was manually reviewed and errors were fixed. Further 12 such iterations were carried with model improvement at every step and annotation accuracy was greater. Transfer Learning was used to improve accuracy despite a smaller dataset. At the end of all iterations a dataset containing 8000 train images and 800 test and validation images was used. This dataset comprises junctions strategically chosen based on model weaknesses at every iteration, with higher focus on locations where the model needed improvement. Data augmentation was used to increase images even for classes like Bicycle that had severe shortage. Overall the entire pipeline was to create auto annotate with current model,review and add data and retrain the model. Metrics like confusion matrix was analyzed while adding more data. Further metrics and training details are discussed in results.

B. Creating Counting Boxes

A script was created that opens up a snapshot of a camera view whose turning pattern needs to be configured, on which we could click the four corners of a region and the coordinates would be logged, making it much easier to create the turning region boxes on our end. The coordinates of the boxes were chosen in a such a way that they cover maximum area thereby decreasing the possibility of a vehicle not being detected. These coordinates are stored in the *config.py* file in the form of a dictionary of camera names mapped to regions of turning boxes.

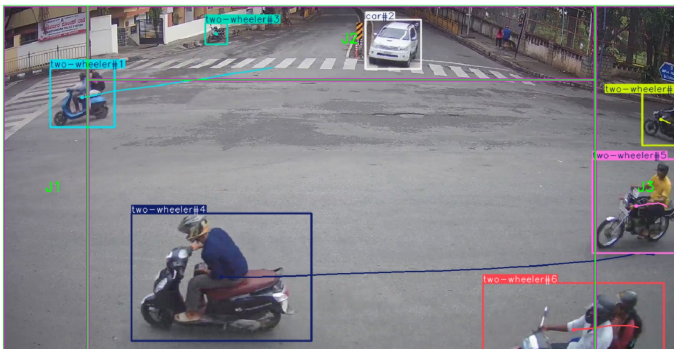

Fig. 2: Comparison of Turning Patterns

C. Counting

The source code of YOLO's own *ObjectCounter* class was modified to implement the turning regions, using the coordinates obtained from *config.py*.Each camera has different regions(boxes) to track movement of vehicle from one box to another. When a vehicle is first detected by the YOLOv8 model, it is assigned a unique ID. When this vehicle is then first detected in a given region/box, it checks for its ID in a global dictionary. If it is not present, we add the unique ID and its corresponding counter box as the source.(Since it was not found in any other box earlier). When the vehicle is detected again in another box, it cross checks for the ID in the dictionary. If present, it adds the box against the ID as a destination box. *config.py* also houses the possible turn patterns for that particular junction, and using this it matches the source and destination boxes to the resulting turn pattern. Taking the *STN_HD_1 junction* as an example, as given in Fig. 2, there are 6 possible turns; BC, BE, DA, DE, FA and FE. The naive approach would be to create six regions and check for turns between them. But when this was tested, it was concluded that many vehicles were making turns while cutting through other regions, which gave wrong counts, eg. when turning from F to C, many vehicles were found going through E and D too. A solution to this is to merge the boxes of each side, and then map the source and destination boxes to turning patterns based on the junction. So here, A and B, C and D and E and F were merged to form three boxes J1, J2 and J3 respectively. Now when a turn was detected with J1 as source and J2 as destination, it would be counted as a BC turn as it is known that AD isn't a valid turning pattern. These mappings are also present in *config.py*.Another addition was in cases,where a rider goes from J1 to J3 through J2 like in Fig.2 or any such similar patterns in other junctions,then only the pattern J3 is considered. In summary, if a vehicle passes through multiple destination boxes, only the final valid destination is counted, as the intermediate ones were logged either due to poor camera position or lane discipline. The entire system is scalable and can be calibrated for new cameras by just defining the boxes and the valid turning patterns.

(a) STN_HD_1 Turning Patterns

(b) Solving the Turning Pattern issue

Fig. 3: Comparison of Turning Patterns

D. Forecasting

A framework for forecasting vehicle counts at various traffic junctions using the using the Auto-ARIMA(AutoRegressive Integrated Moving Average) model was developed. Given the dynamic nature of Indian roads with high variability from location to location and other factors like time, a time series model with fixed hyper parameters was not used. Auto ARIMA was used to dynamically find the optimal parameters for forecasting depending on the data logged,thereby increasing accuracy considerably.Auto-ARIMA automates the process of finding the best combination of ARIMA (AutoRegressive Integrated Moving Average) model parameters (p, d, q) by evaluating multiple combinations and selecting the one that minimizes a given information criterion (such as AIC or BIC), ensuring the model is both parsimonious and accurate. The preprocessing phase includes organizing and interpolating vehicle count data

to match with set time intervals, ensuring consistent data across different time periods. The frame number was logged at all destination when a vehicle made a particular turning pattern. This frame number was key in preprocessing the data. The auto-ARIMA model was applied to each vehicle class across multiple traffic turns, with hyper parameters optimized using both grid search and random search techniques. The model was trained on a 30-minute historical data to predict counts for the next 30 minutes, with predictions being compared to actual observations for accuracy. A function was configured where the deviation was calculated as per the metrics to be used by the organisers, to provide a better view of how the model was performing. And other metrics such as MAE and RMSE was used along with visualization of results to find the best paramters for the Auto ARIMA model.

IV. Explanation

This section covers each file in detail, explaining the work done in the modular code.

A. Directory Structure

The directory structure is organized as follows:

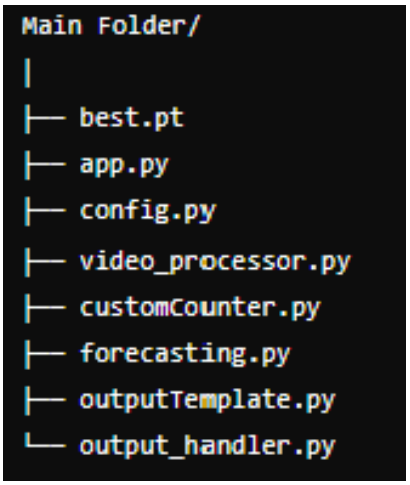

Fig. 4: Directory Structure

B. File Overview and Functionality

Fig. 5: Execution Flow Overview

1) app.py:

- `main` function processes video data from a JSON input.
- For each camera ID and its associated videos:
  - **Counting Vehicles:** Calls `process_videos_and_compile_counts` to obtain cumulative counts and frame count dictionary.
  - **Forecasting Vehicle Counts:** Uses `forecast_vehicle_counts` to predict future vehicle counts.
  - Updates results with camera ID using `update_counts_with_camera_id`.
- Saves the results as a JSON file to the specified output file.

2) config.py: This script houses a dictionary with the valid turning patterns and junction coordinates for each of the camera views. Separating this from other has made the code neat and is highly scalable.

3) video_processor.py:

- **Turning Patterns:** `turning_patterns` is a dictionary where each key is a pattern (e.g., `'AB'`) and each value is a tuple (src, dst) indicating source and destination junctions.
- **Count Updates:** `count_dict` and `frame_count_dict` are updated based on detected vehicles and their movements between regions as per the defined patterns.

4) customCounter.py:

- **Attributes:**
  - `names`: Dictionary of class names for objects.
  - `reg_pts`: Points defining the counting region (line or polygon).
  - `line_dist_thresh`: Distance threshold for line-based counting.
  - `count_ids`: Dictionary tracking object IDs and their states.
  - `class_wise_count`: Counts of objects entering and exiting based on class.
  - `track_history`: History of tracked object positions.
- **Counting Region:** Defined by `reg_pts`, which can be a line (`LineString`) or a polygon (`Polygon`).
- **Distance Threshold:** `line_dist_thresh` is used to determine if a tracked object is close enough to a line for counting.
- **Tracking History:** Maintained for each object ID to track its movement and determine if it has crossed the counting region.

5) forecasting.py:

- The `Forecaster` class handles vehicle count forecasting using ARIMA models.
- It saves frame counts to a dataframe and reads the data for analysis.
- The ARIMA model is used for forecasting, with results returned for each vehicle class.
  - **Model Selection**: `auto_arima` is used to automatically select the best ARIMA parameters.
  - **ARIMA Parameters**:
    - `start_p=0`: Starting value for the auto ARIMA model's autoregressive order.
    - `start_q=0`: Starting value for the auto ARIMA model's moving average order.

    - `max_p=5`: Maximum value for the autoregressive order.
    - `max_q=5`: Maximum value for the moving average order.
  - **Forecast Points**: Set to 120, representing a 30-minute forecast period.
  - **Error Handling**: If model fitting fails, the forecast defaults to zero.

6) outputTemplate.py:

- **Top-Level Key (`Cam_ID`)**: Represents a specific camera ID.
- **Second-Level Key (`Cumulative Counts`)**: Holds cumulative counts for different traffic patterns.
- **Pattern Keys (`AB`, `BA`, etc.)**: Each key corresponds to a unique traffic pattern or scenario.
- **Vehicle Count Data**: For each pattern, counts are initialized for various vehicle types including Bicycle, Bus, Car, LCV, Three Wheeler, Truck, and Two Wheeler, all set to zero.

7) output_handler.py:

- **Initialization**: Uses the `template` from `outputTemplate` to create a new dictionary for a given camera ID.
- **Output**: Returns the updated dictionary with cumulative and predicted counts.

V. RESULTS

The final model was trained for 350 epochs with features like dropout, early stopping and label smoothing to account for bounding box inaccuracies. The YOLOv8 model achieved a precision of 0.909 and recall of 0.875 while classes like cars and buses achieved a precision of 0.95. The overall pipeline achieved a deviation of 20.79% for counting and 28.41% deviation in forecasting at its best.

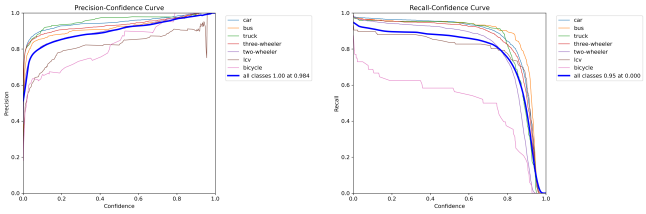

Fig. 6: P curve (left) and R curve (right)

VI. CONCLUSION

This project successfully covers a modular and scalable system for analyzing and predicting vehicle counts by class across multiple camera locations. The framework is designed such that it works despite challenging Indian road conditions like poor lane discipline by combining junctions,thus increasing accuracy. The system effectively processes video inputs, counts vehicles, and forecasts the dynamic future traffic patterns using an Auto-ARIMA-based model.

Overall, this system provides a robust framework for real-time traffic monitoring and forecasting, which can be adapted to different environments and expanded with additional features or more sophisticated models. The approach taken in this project serves as a strong foundation for further research and development in traffic analysis and prediction.

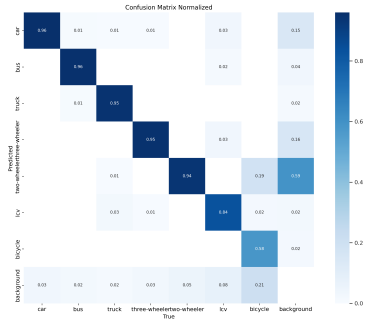

Fig. 7: Confusion matrix

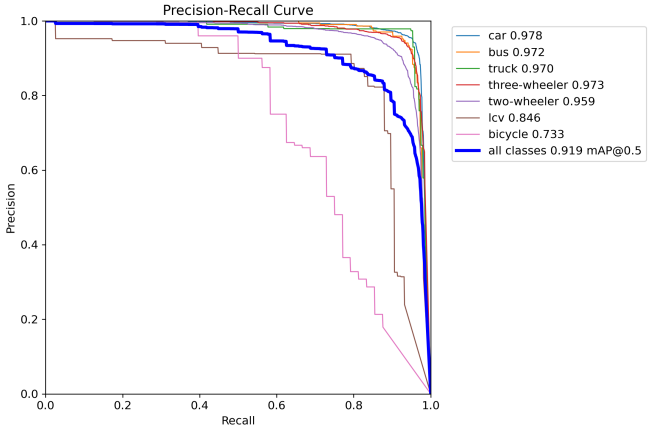

Fig. 8: PR curve

A. Shortcomings of our model

- If a vehicle overlaps and covers another vehicle, there are chances of the overlapped vehicle going undetected.
- The forecasting relies heavily on historical data, and anything that deviates from the general trend is likely to go unforecasted. For example, a road blockage due to VIP movement, or an accident.

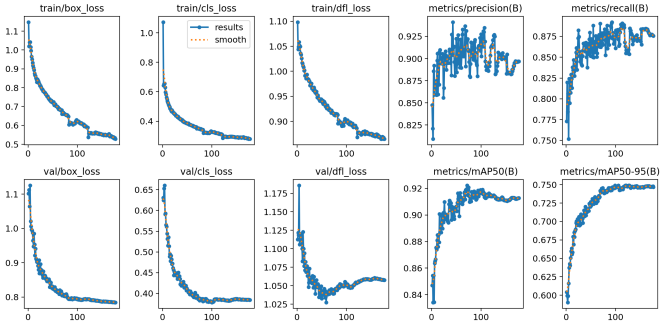

Fig. 9: Results

REFERENCES

[1] Irhami, E. A., & Farizal, F. (2021). Forecasting the number of vehicles in Indonesia using Auto Regressive Integrative Moving Average (ARIMA) method. Journal of Physics Conference Series, 1845(1), 012024. https://doi.org/10.1088/1742 6596/1845/1/012024

[2] Zhang, Y. (2020). Short-Term Traffic Flow Prediction Methods: A survey. Journal of Physics Conference Series, 1486(5), 052018. https://doi.org/10.1088/1742-6596/1486/5/052018

[3] Lee, J., Lee, S., Choi, H., & Cho, H. (2021). Time-Series data and analysis software of connected vehicles. Computers, Materials & Continua/Computers, Materials & Continua (Print), 67(3), 2709–2727. https://doi.org/10.32604/cmc.2021.015174

[4] Lippi, M., Bertini, M., & Frasconi, P. (2013). Short-Term Traffic Flow Forecasting: An experimental comparison of Time-Series analysis and Supervised learning. IEEE Transactions on Intelligent Transportation Systems, 14(2), 871–882. https://doi.org/10.1109/tits.2013.2247040

[5] YOLOv8 - Jocher, G., Chaurasia, A., & Qiu, J. (2023). Ultralytics YOLO (Version 8.0.0) [Computer software]. https://github.com/ultralytics/ultralytics

[6] Tzutalin. LabelImg. Git code (2015). https://github.com/tzutalin/labelImg