Pavement Eye: Automated Pavement Condition Index (PCI) Assessment Using YOLO-based Crack Detection and Geolocation Analysis

Authors

Yahya M. Zakaria[1], Doaa A. Farid[1], Rowan H. Mohamed[1], Salsabel O. Abd Elhafiez[1], Mohamed G. Abdelzaher[2]

[1] *Computing and Data Science, Faculty of Computers and Data Science, Alexandria University, Egypt*
[2] *Computers and Systems, Faculty of Engineering, Alexandria University, Egypt*

cds.YahiaMahmoud24700@alexu.edu.eg, cds.DoaaAbdelsattar250127@alexu.edu.eg, cds.RawanHussien250141@alexu.edu.eg, cds.SalsabilOsama250176@alexu.edu.eg, Es.mohamed.gomaa2023@alexu.edu.eg

1 ABSTRACT

This paper discusses the implementation of "Pavement Eye" from technical and scientific perspective rather than business one. Here we will discuss things like the calculations of Pavement condition index(PCI). Technologies used in the pipline and why they are used ?. Deep learning models. Challenges faced. How to improve in the future.

INTRODUCTION

Pavement Eye is an intelligent system designed to automatically detect the location and types of pavement cracks using deep learning techniques in a cost-effective and efficient manner. The system leverages advanced data engineering practices to build a high-performance, real-time data streaming pipeline, ensuring rapid processing and analysis of large volumes of roadway imagery. Furthermore, the application integrates seamlessly with OpenStreetMap (OSM), enabling enhanced spatial analysis and visualization of pavement conditions within a broader geographic context.

Data Pipeline and technologies

We developed a scalable streaming data pipeline to collect a large volume of crack data efficiently, leveraging the widespread use of mobile devices during driving. This approach enables continuous data acquisition from diverse environments, making the system more adaptable and representative of real-world road conditions.

Flutter

Automatically captures image every specificed time interval for example (1 second) and sends the data to the backend. It also sends other information in json format. The application sends image, location (Latitude, Longitude), and Pixel per meter(ppm).

User can specify his settings such as:

- Stop or start camera.
- Frequency of images taken in seconds.
- Pixel per meter based on camera quality.
- URL of the backend.
- Reset settings.

The application also shows the status of requests, last taken image, and live camera stream for better debugging.

Backend using Flask

The backend receives requests from the Flutter application using websockets(this optimization was made in the development and this enhanced the application and reduced bandwidth) and loads the deep learning model (details discussed in a later section). If the submitted image contains only cracks, the system stores the image in the data lake and sends the associated metadata to Kafka, initiating the data processing pipeline. The backend operates outside docker environment (for easier coding, can be converted to docker container in deployment). You can see what is send from flutter to the backend in table [1].

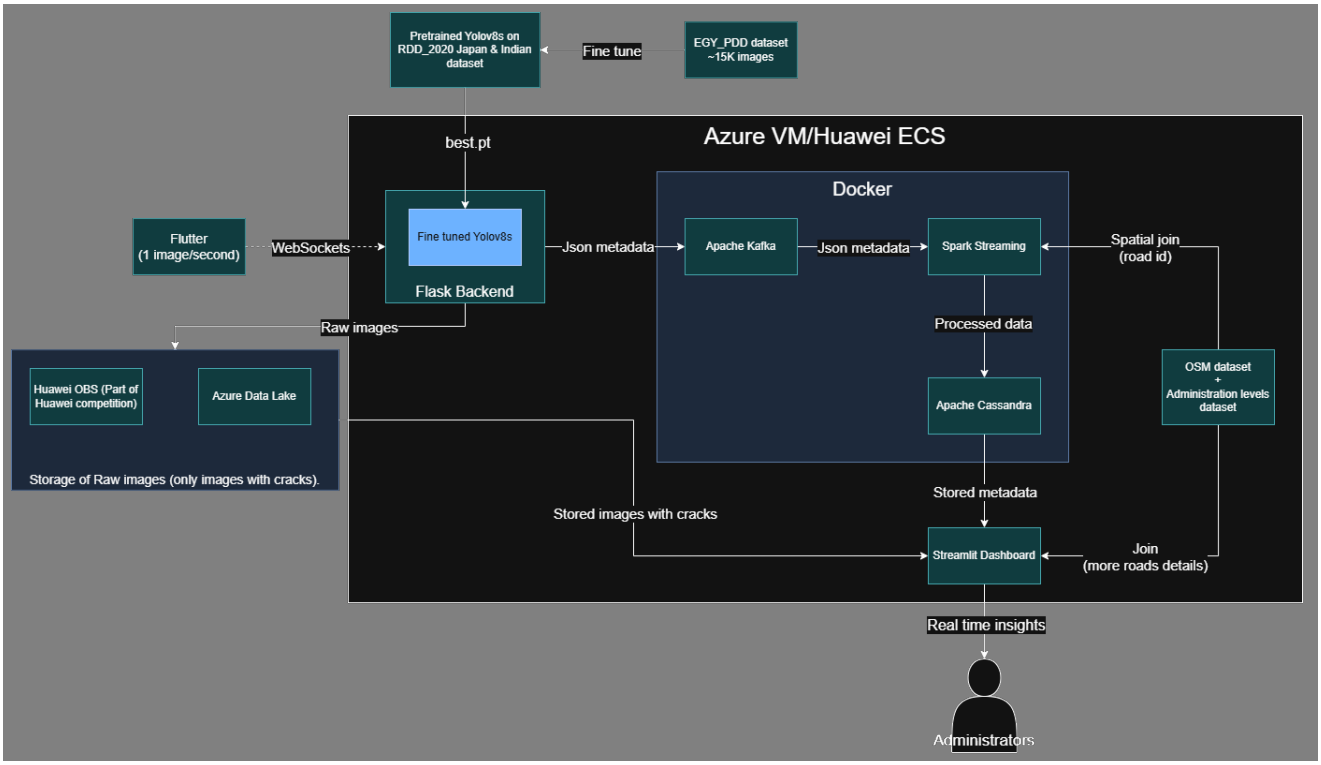

Fig. 1. Full pipeline

Apache Kafka

The system receives data from the backend in JSON format. Kafka acts as a bridge, transferring data from the Flask websockets to Spark Streaming. It provides a scalable and fault-tolerant solution, ensuring reliable data delivery. In case of failures, Kafka persists data on a durable file system to prevent loss. You can see what is send from backend to apache kafka in table [2].

Table 1. Features sent from backend to the Kafka in JSON format

Name	Description
`lon`	Longitude where the image was taken
`lat`	Latitude where the image was taken
`time`	Local timestamp when the image was taken (ISO 8601 format)
`labels`	List that contains a dictionary for each detected crack
`ppm`	Pixels per meter; used to calculate the Pavement Condition Index (PCI)
`image`	Image filename in the cloud storage in format `lon_lat_timestamp.jpg`

Table 2. Items in labels feature in table

Name	Description
`label`	Type of crack, ex: Alligator, Pothole, etc.
`confidence`	Confidence of model's detection
`x1, x2, y1, y2`	Bounding box

Spark streaming

It is the main processing engine in the pipeline. It takes the streaming data from Apache Kafka. And perform the following pre-processing:

- Parse JSON data and organize the schema into a data frame.
- Removing nulls.
- Check for corrected data such as valid ranges for some columns.
- Validate data types.
- Spatial Join with OSM to specific street, government, and district of the crack from its location with the help of Geopandas library.
- stores data in Apache Cassandra.

We chose Spark Streaming for its scalability, fault tolerance, seamless Kafka integration, low-latency processing, and rich support for complex data transformations and spatial joins.

Apache Cassandra

We chose Cassandra because it is well-suited for write-intensive applications, such as those in the IoT domain. Our goal is to scale the application to handle thousands of crack detections by many concurrent users. Cassandra's architec-

ture is optimized for high write throughput and low latency, making it an ideal choice to support this level of concurrent data ingestion. We only created one table as it is the bes case for cassandra and put district as a partition key as we will filter on it in the future. You can see the table schema in table [3].

Streamlit

We performed data analysis and created interactive visualizations to gain better insights. For this purpose, we utilized libraries such as Pydeck for interactive maps, along with Matplotlib and Plotly for various plotting needs.

The analyzed data originates from Apache Cassandra and is joined with the OpenStreetMap (OSM) dataset using the `road_index` field generated during the spatial join step in Spark Streaming. This approach significantly improves performance by avoiding expensive bulk spatial joins, which are computationally intensive. We ensured also context and storytelling approach by relating data to each other and integration with OSM.

Question this dashboard can answer:

1. What are street that needs to be repaired first (most damaged), (lower PCI) ?
2. What is the region/district that have the largest number of cracks ?
3. What is the distribution of crack types in top 10 damaged roads ?
4. Is maximum allowed speed affects road damage, and How much ?
5. Is the structure of roads (bridges/tunnels) affects road damage, and How much ?
6. How much improvement/declinement in roads quality in general in the last 10 months ?
7. Distribution of cracks across different road types (Primary, link, residential, etc).
8. Is this crack real ? (validation method to check if the detection is correct from the stored images and bounding boxes).

Table 3. Schema of crack table

Column Name	Data Type	Description
`id`	`uuid`	Unique identifier for each crack record
`road_index`	`int`	Index representing the road segment
`timestamp`	`timestamp`	Date and time when the crack data was recorded
`confidence`	`float`	Confidence score of the crack detection
`image`	`text`	Filename or path of the associated image
`lon`	`double`	Longitude coordinate of the crack location
`lat`	`double`	Latitude coordinate of the crack location
`x1, y1`	`double`	Top-left coordinates of the bounding box
`x2, y2`	`double`	Bottom-right coordinates of the bounding box
`ppm`	`double`	Pixels per meter ratio for PCI calculation
`dist`	`text`	region identifier (partition key) (such as Montaza)
`label`	`text`	Type of crack detected (e.g., Alligator Crack)

Docker

To simplify environment setup and streamline future deployment, we used Docker to containerize three key components: Spark, Kafka, and Cassandra.

Cloud technologies

We utilized Azure Virtual Machines (Huawei ECS as part of the Huawei Spark competition) to host and deploy all project components. Meanwhile, Azure Data Lake (Huawei OBS) was used for storing raw image data. This setup allows us to efficiently combine raw images with predicted bounding boxes from the dataset, enabling future model fine-tuning and continuous improvement in prediction accuracy.

Moreover, this architecture provides a fast, scalable, and efficient data collection pipeline, paving the way for building large-scale datasets that can support future research and development. Leveraging cloud technologies ensures that our system remains scalable, fault-tolerant, and adaptable as the project evolves and grows.

Ngrok

Used in testing to use on of our Laptops as a server. We published the backend on port 5000.

Deep learning model

We initially used the YOLOv8s model trained on the RDD_2022 dataset. However, applying this model directly led to noticeable issues due to the complex and unique nature of Egyptian roads. Although the RDD_2022 dataset was designed for international use and includes images from countries such as India and Japan, it did not generalize well to the environmental and infrastructural conditions of Egyptian roads. You can download the weights we used from:
https://github.com/oracl4/RoadDamageDetection

To address this limitation, we fine-tuned the pretrained model on the EGY_PDD dataset [1], which is currently the only publicly available dataset for pavement distress detection in Egypt. This fine-tuning significantly improved the model's performance by allowing it to better adapt to local road conditions. Furthermore, it expanded the model's

detection capabilities from the original four crack types to eleven, adding seven new distress categories specific to Egyptian pavement characteristics.

Contact:
yahiamhamoood333@gmail.com To access model weights

EGY_PDD dataset [1] was collected in a manner that very similar to our case and how our pipeline will be in the future. The fine tuning was done on Laptop RTX 3050 Ti with 4 VRAM in nearly 7.5 hours on all the dataset nearly 15000 images for 64 epochs. You can see model labels in table [4].

The speed of the model itself is 7.2 ms per image. with mAP50 of 0.55 which is better than the models of the dataset creators(descriped in the paper).

OSM integration

OpenStreetMap (OSM) is a collaborative project that creates a free, editable, and detailed map of the world. It is built by a community of mappers who contribute and maintain geographic data such as streets, buildings, landmarks, and natural features. OSM data is openly available and widely used in research, navigation, and geographic information systems (GIS) applications due to its richness, flexibility, and constantly updated content. The features that we used from OSM dataset is described in table [5].

We get the data from Geofrabrik which is a german organization that collects data from OSM you can find in the link:
https://download.geofabrik.de/africa/egypt.html

These additional data significantly enhanced our solution by enabling deeper analysis and understanding of the correlations between crack types and the Pavement Condition Index (PCI) across various road types and structures—such as tunnels and bridges—under different conditions, including traffic speed.

Other external data

We also used additional datasets that define administrative boundaries, such as districts and governments. This allows us to identify the exact administrative area where a crack is located and enables users to filter results by district or government. It also improves system performance by allowing data partitioning based on districts.

You can get the dataset from: https://data.humdata.org/dataset/cod-ab-egy

Table 4. Distress types in the EGY_PDD dataset

Distress Type	Label (EGY_PDD)
Rutting	D0
Reflective & Transverse Cracks	D1
Block Cracks	D2
Longitudinal Cracks	D3
Alligator Crack	D4
Patching	D5
Potholes	D6
Bleeding	D7
Corrugation	D8
Raveling & Weathering	D9
Bumps & Sags	D10

Calculation of PCI

Pavement Condition Index (PCI) is a standardized numerical metric ranging from 0 to 100 that quantifies the overall condition of a pavement surface. A PCI value of 100 denotes a pavement in excellent condition, whereas a value of 0 indicates a pavement that has failed completely. The PCI is derived from systematic visual inspections that assess the types, severities, and extents of surface distresses, such as cracks and potholes. This index is extensively utilized by transportation agencies to guide maintenance prioritization and rehabilitation planning.

Steps used to calculate PCI from features we have

Cacluate crack area

Given that our dataset consisted solely of images annotated with bounding boxes, we utilized these annotations to estimate the physical dimensions—specifically, the length and width—of the objects of interest. These dimensions were initially measured in pixels, as derived directly from the bounding box coordinates.

$$CrackWidth(m) = \frac{|x_2 - x_1|}{ppm} \tag{1}$$

$$CrackLength(m) = \frac{|y_2 - y_1|}{ppm} \tag{2}$$

$$CrackArea(m^2) = CrackWidth \times CrackLength \tag{3}$$

Cacluate road area

We made an assumption here that all roads have static width that equals to 10 meters. We did this because the OSM includes the length of the roads that we can get

Table 5. Feature that OSM adds in the project

Name	Description
`name`	Name of the road
`oneway`	Is this a oneway road? “F” means that only driving in direction of the linestring is allowed. “T” means that only the opposite direction is allowed. “B” (default value) means that both directions are ok.
`maxspeed`	Max allowed speed in km/h
`bridge`	Is this road on a bridge? (“T” = true, “F” = false)
`tunnel`	Is this road in a tunnel? (“T” = true, “F” = false)
`geometry`	Shape represents the road

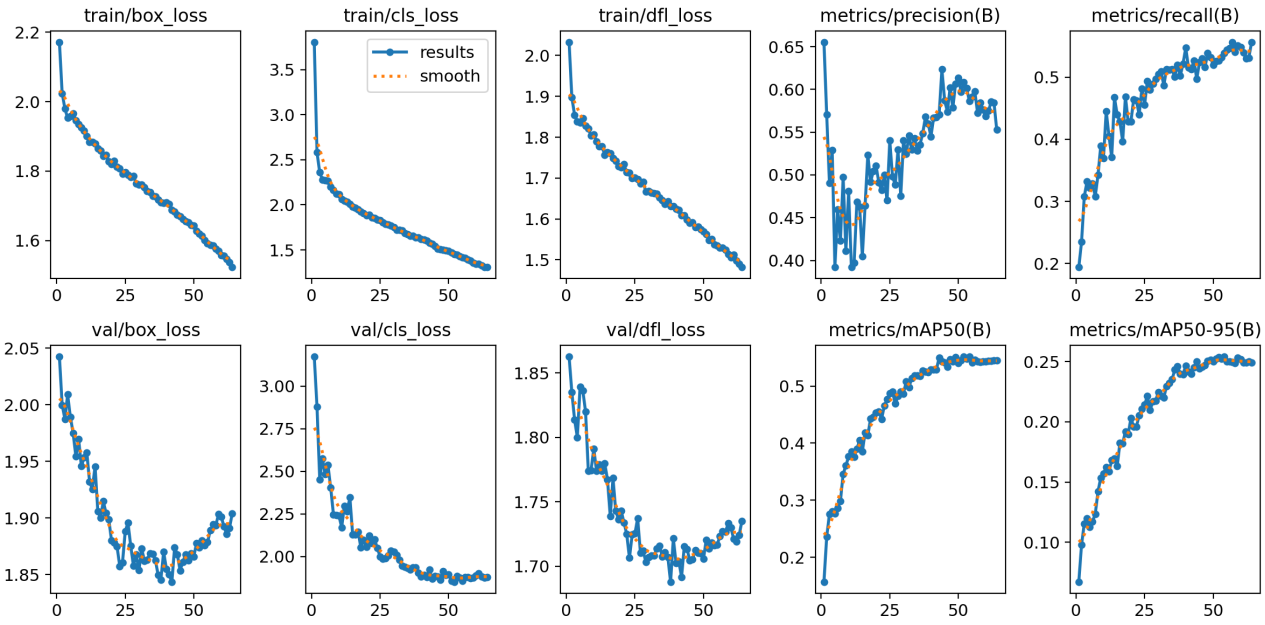

Fig. 2. Evolution vs epochs

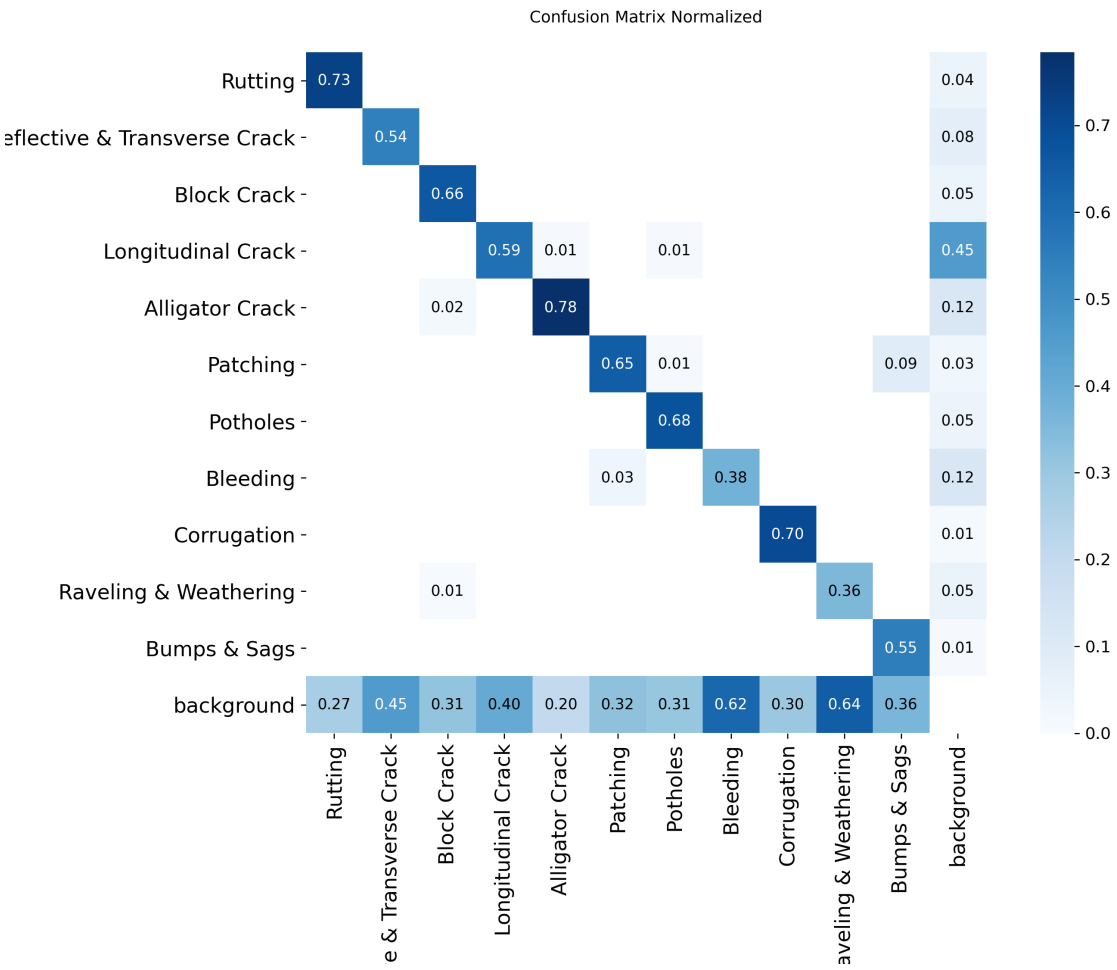

Fig. 3. Normalized confusion matrix of the fine tunning phase on yolov8s model

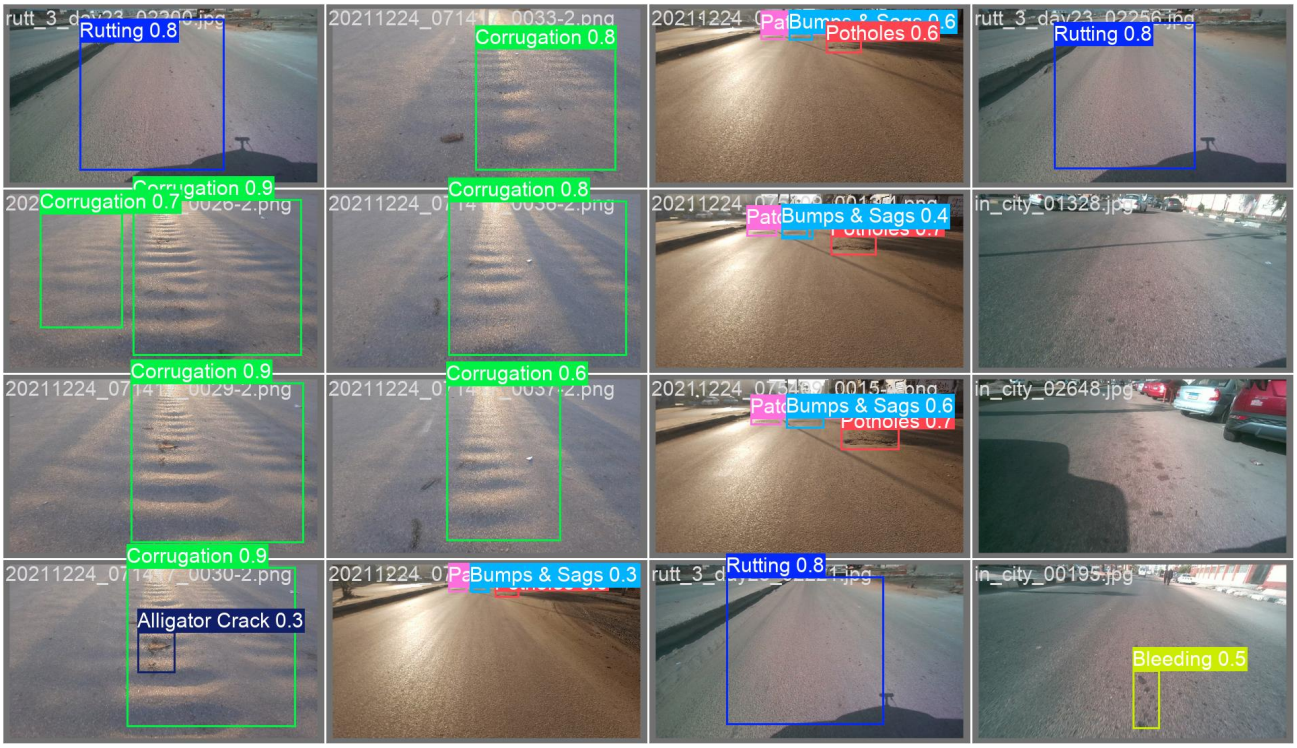

Fig. 4. Some of predicted objects

from geometry shapes. As it was in *LineString* format not *Polygon* we can not get width.

Assuming a uniform road width $W_{\text{road}} = 10$ meters, the road area is:

$$A_{\text{road}} = L_{\text{road}} \times W_{\text{road}} \tag{4}$$

Then we can easily calculate distress density (DD_{ij}) for crack i in road j by

$$DD = \left(\frac{A_{\text{crack}}}{A_{\text{road}}} \right) \times 100 \tag{5}$$

A *deduct value* is assigned based on the distress type and its density using a function:

$$DV = f(\text{label}, DD) \tag{6}$$

This function is derived from expert knowledge or standards such as ASTM D6433. There is different function for each crack type.

Let $\{DV_1, DV_2, \ldots, DV_n\}$ be the list of deduct values for a road segment.

Case 1: No cracks

$\text{PCI} = 100$

Case 2: One defect

$\text{PCI} = 100 - DV_1$

Case 3: Multiple defects

We compute the **Total Deduct Value (TDV)** as:

$$TDV = \sum_{i=1}^{n} DV_i$$

Then apply a correction to compute the **Corrected Deduct Value (CDV)**:

$$\text{CDV} = \begin{cases} TDV - \dfrac{TDV^2}{250}, & \text{if } TDV \leq 100 \\ 100 - 10 \cdot \sqrt{TDV - 100}, & \text{if } TDV > 100 \end{cases}$$

Finally, the PCI is:

$$\text{PCI} = \max(0, 100 - CDV)$$

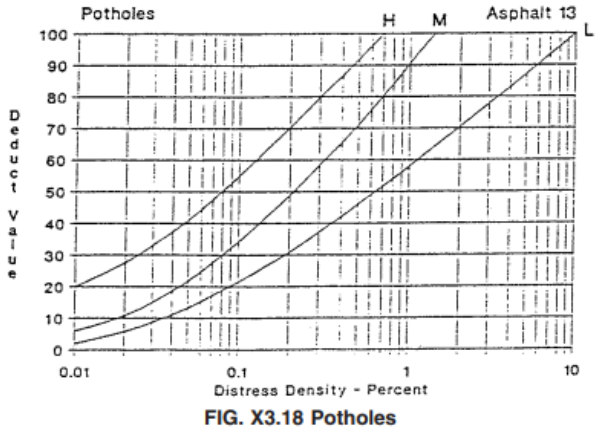

Fig. 5. Potholes function

PCI Condition Classification

The final PCI value is classified into a condition category as follows:

$$\text{Condition} = \begin{cases} \text{Excellent}, & \text{if PCI} \geq 85 \\ \text{Good}, & \text{if } 70 \leq \text{PCI} < 85 \\ \text{Fair}, & \text{if } 55 \leq \text{PCI} < 70 \\ \text{Poor}, & \text{if } 40 \leq \text{PCI} < 55 \\ \text{Very Poor}, & \text{if } 25 \leq \text{PCI} < 40 \\ \text{Failed}, & \text{if PCI} < 25 \end{cases}$$

Challenges and Future Work

During the development of the system, two main challenges were encountered:

1. **Real-time Processing:** Achieving high-speed performance suitable for streaming data was a significant challenge. This was addressed by migrating the communication protocol from a traditional RESTful API to **WebSockets**, enabling continuous, low-latency data transmission.
2. **Complex Nature of Egyptian Roads:** The heterogeneous and irregular characteristics of Egyptian roads posed difficulties for accurate distress detection. To overcome this, the model was **fine-tuned on the EGY_PDD dataset** [1], which is specifically designed for Egyptian pavement conditions.

Future Improvements: Several aspects can be further optimized in future work, including:

- Developing a more precise method for **Pavement Condition Index (PCI)** calculation.
- Integrating **edge computing devices** to reduce backend processing load and improve scalability.
- Use **Agentic AI** for faster and easier insights.

CONCLUSION

This paper presented *PavementEye*, an end-to-end system for automated pavement condition assessment that combines YOLOv8s-based crack detection with a scalable streaming data pipeline, spatial enrichment using OpenStreetMap, and PCI computation for actionable maintenance insights. We demonstrated a practical architecture that leverages mobile data collection, Kafka and Spark for reliable real-time processing, Cassandra for high-throughput storage, and Streamlit-based visualizations for exploratory analysis. While the prototype successfully detects and geolocates common distress types and produces PCI estimates, we also identified important limitations related to model generalization, sensor quality, environmental variability, and real-time constraints.

To address these gaps we proposed a set of targeted enhancements—ranging from domain-specific model retraining and multi-sensor fusion to cloud hybrid processing and predictive maintenance modules—which together form a clear roadmap toward robust, city-scale deployments. Moving forward, validating the system on larger, region-specific datasets and piloting integration with municipal maintenance workflows will be essential to quantify operational benefits and refine the tool for real-world use. In sum, PavementEye offers a promising, cost-effective approach to improve pavement monitoring and enable more data-driven infrastructure stewardship.

0 REFERENCES

[1] M. F. Abdelkader, M. A. Hedeya, E. Samir, A. A. El-Sharkawy, R. F. Abdel-Kader, A. Moussa, and E. El-Sayed, "Egy_pdd: a comprehensive multi-sensor benchmark dataset for accurate pavement distress detection and classification," *Multimedia Tools and Applications*, vol. 84, pp. 38509–38544, Sep 2025.

THE AUTHORS