Industry Project Report

David Shamess `ds18sa@brocku.ca`

Ethan DeSantis `ed21dc@brocku.ca`

Hansel Janzen `jh23px@brocku.ca`

Alexandre Brillon `la23nr@brocku.ca`

December 19, 2025

Abstract

This project presents a proof of concept intelligent navigation system that integrates computer vision, graph based pathfinding, and geospatial visualization to support safety aware route planning. A custom trained YOLOv11x object detection model is used to identify road hazards from street level imagery, including potholes, debris, logs, and construction related elements. Detected hazards are translated into weighted safety scores that augment a graph representation of an urban road network, enabling routing decisions that account for both distance and risk.

The system architecture combines local image inference, a Flask based backend, JSON based data storage, and an interactive Leaflet/Folium web interface for visualization and user interaction. Road segments are modeled as weighted edges whose safety scores are dynamically updated based on detected hazards.

Through iterative model development, dataset expansion, and refined hazard class definitions, detection performance was significantly improved, demonstrating the importance of data quality and class consistency in applied computer vision systems. Overall, this project illustrates the feasibility and potential of integrating object detection with geospatial analytics to enhance navigation systems with contextual safety information, while providing a scalable foundation for future real time deployment.

1 Data Collection

The image dataset used to train our YOLO based road hazard detector was sourced from Mapillary, a platform that hosts crowdsourced street level imagery. We selected a region centered on (44.89814752321598, -76.24493951025102), in Perth, Ontario. This location was chosen because (1) the road network has a grid like structure that is easy to model as a graph for routing, and (2) Mapillary had an abundant amount of imagery coverage in this area, enabling consistent visual input for our computer vision model.

To collect images, we implemented a Python script (boxgetimg.py) that queries the Mapillary API using the selected geographic region and downloads available images along with associated metadata (e.g., latitude/longitude). Using this process, we retrieved over 3,000 candidate images from the target area. After an initial review, we found that the majority of images did not contain visible road hazards relevant to our target hazard classes, which would limit our ability to train a hazard detection model within the project timeframe.

To address this, we generated an augmented training set using a generative image editing workflow. Starting from real Mapillary road images, we created modified variants that (a) introduce environmental variation (e.g., day/night, rain, snow) and (b) insert one hazard instance from our target set: pothole, construction elements, fallen log, or debris. We produced over 1,000 augmented images for annotation and model training.

2 Annotation Process and Class Definitions

All images were annotated using Label Studio, a tool that supports bounding box labeling for object detection tasks. The annotation process focused on identifying specific road hazards. Initially, the classes we identified as hazards include Construction, Logs, Debris, Snow, and Potholes. Bounding boxes were drawn tightly around individual hazard objects to ensure precise localization and accurate training labels, but it was challenging to draw bounding boxes over "objects" that are amorphous like snow or debris. After encountering some challenges while training our model (Section 6), our final product was trained on the following hazard categories: Pylon, Barrier, Barrel Debris, Pothole, Log.

Bounding box annotation made it challenging to identify snow and debris, but it was chosen over full scene or environment level classification because the goal of the system was to perform risk assessment based on individual hazards present on a road. By detecting and classifying specific objects, the system can assign weighted safety scores based on the nature and frequency of hazards present in an image.

3 Geospatial Mapping Methodology

Mapping out the real life location began by creating a graph to simulate the covered intersections our model would be "watching". Nodes in the graph represented intersections, and stored the associated longitude and latitude coordinates of the intersection, as well as a label of the street names that they intersected at. Edges represented a connection of intersections, and stored image data, safety score data, and the distance between nodes, calculated using Haversine distance to be used in calculating the shortest route. Creating the visual overlay of of the graph representing the real world location the graph is modeled after was implemented in the final system using folium. Folium is a python library for creating web app maps.

To represent the real world road network of our selected area, we modeled the location as a weighted graph $G = (V, E)$. Each node $v \in V$ represents a road intersection and stores (1) a human readable intersection label (street names) and (2) the geographic coordinates (latitude, longitude). Each edge $e = (u, v) \in E$ represents a road segment connecting two adjacent intersections. Edges store attributes required by the system pipeline: (a) a list of associated street level images for that road segment, (b) the current safety score (updated as new detections are produced), and (c) the physical distance between intersections.

To visualize the road network and model coverage, we generated an interactive web map using Folium (a Python library that produces Leaflet.js maps as HTML). The map renders intersections as markers and road segments as polylines drawn between node coordinates. Road segments are styled according to their safety score (e.g., green = low risk, orange = moderate risk, red = high risk), enabling a visual overlay of road conditions. Edge details for a route are displayed such as safety score, and an image if available for that segment.

4 AI Model Development and Fine Tuning

4.1 Model Selection

The computer vision model uses YOLOv11x, selected for its strong accuracy on object detection tasks and its practicality for local inference in a prototype development environment.

We fine tuned the model using a standard transfer learning workflow (starting from pre-trained weights and adapting the pretrained model to our road hazard classes), using a Dell 5860 designer machine equipped with a Nvidia RTX A4000 for model training.

4.2 Dataset Format and Training Pipeline

All annotations were exported in YOLO format (one label file per image containing class IDs and normalized bounding box coordinates). This enabled a straightforward Ultralytics training pipeline using a standard dataset structure (train/val image folders with corresponding label folders). Images without hazards were included as negative/background examples (i.e., images containing no labeled objects).

To train the model, we followed a step by step guide by EJ Technology Consultants[3], which provided a practical reference for dataset structure, training invocation, and interpreting training outputs.

4.3 Iterative Model Development

We developed two primary model versions.

4.3.1 Model 1: Baseline Detector

Model 1 was trained as an initial baseline using a smaller dataset with fewer than 500 images containing labeled hazards. The training split contained 834 images (including 369 images with hazard labels and the remaining images as negative/background). The validation split contained 237 images (including 34 images with hazard labels and the remaining images as negative/background). Model 1 was trained to detect the following classes:

- Construction
- Logs
- Debris
- Snow
- Potholes

The model was trained for 60 epochs. Overall, Model 1 showed early signs of detection capability but performance was inconsistent: it occasionally detected potholes but missed others in the same scene, performed poorly on debris, and showed weak or absent detection for logs. Construction detection was also inconsistent. These outcomes suggested the baseline model did not reliably converge to a robust solution under the initial dataset size/class definitions.

4.3.2 Model 2: Improved Detector (Expanded Data + Revised Classes)

To improve on the limitations of Model 1, we trained Model 2 using an expanded dataset of 1125 total annotated images, 120 epochs and revised class definitions to improve object

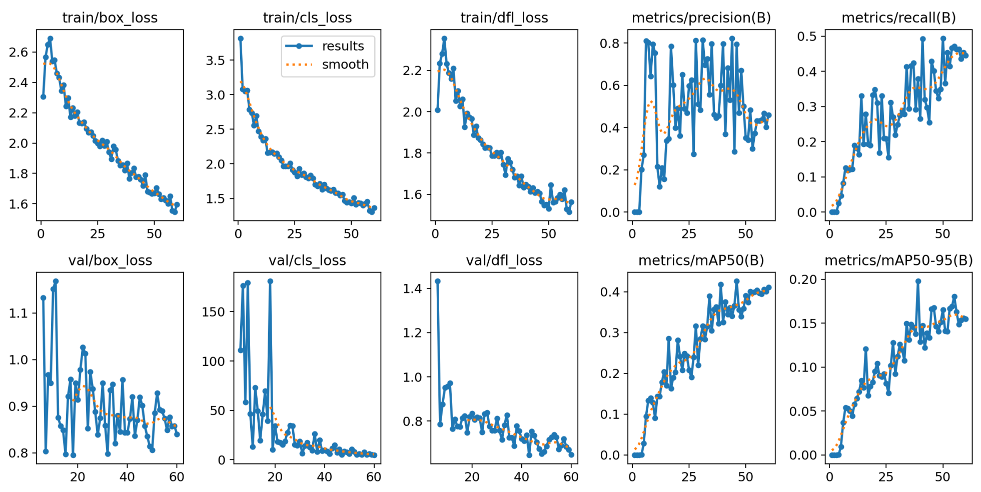

Figure 1: Training Curves / Results Summary - Model 1

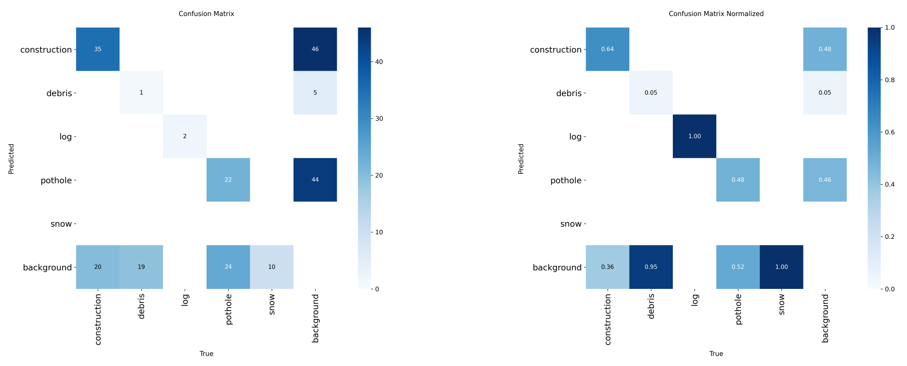

Figure 2: Confusion Matrices (regular, normalized) - Model 1

visual consistency. The dataset was split 70/30 into training and validation sets.

A key refinement was the change in classes, splitting the broad construction category into more visually consistent subclasses.

- Debris
- Log
- Pothole
- Pylon
- Barrier
- Barrel

For simplicity and to reduce class ambiguity, we removed the Snow category. Using this newly annotated dataset and updated class set, Model 2 achieved improved detection accuracy and more reliable identification of hazards relevant to safety scoring.

4.3.3 Model 2: Training Results and Graph Analysis

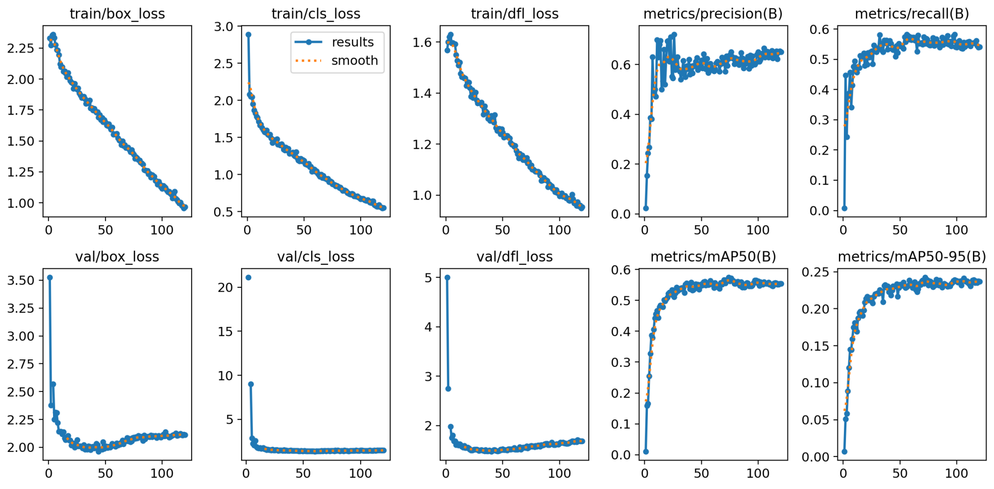

Figure 3: Training Curves / Results Summary - Model 2

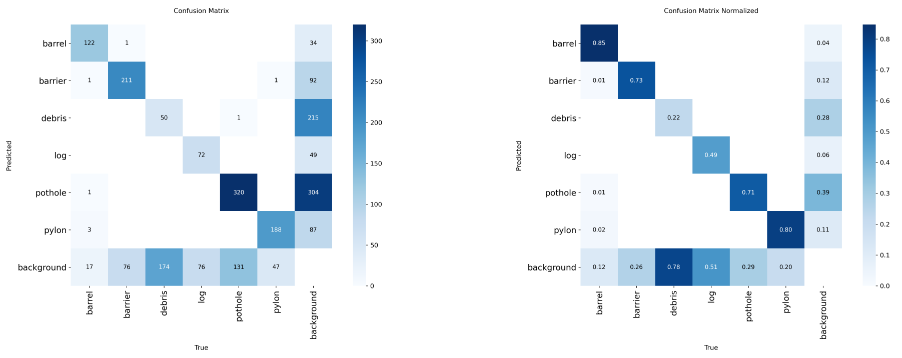

Figure 4: Confusion Matrices (regular, normalized) - Model 2

Training and Validation Curves (`Figure 3`) The results summary for Model 2 shows substantially more stable learning behavior than Model 1. Training losses (box loss, classification loss, and DFL loss) decrease smoothly over the full training run (120 epochs), indicating improved localization and classification as training progresses. Validation losses drop sharply early on and then largely stabilize; toward later epochs, the validation box/DFL losses trend slightly upward while mAP remains mostly flat, which suggests diminishing returns (and mild overfitting) after the model has largely converged. Metrics stabilize at a higher level than Model 1, with precision and recall curves becoming smoother and mAP 50 plateauing near ≈ 0.56, consistent with the precision recall curve summary.

Confusion Matrices (`Figure 4`) The confusion matrix compares true labels (columns) to predicted labels (rows). Model 2 has a strong diagonal for several classes (e.g., Barrel 122 correct, Barrier 211, Pothole 320, Pylon 188), indicating improved class separability after refining Construction into visually consistent subclasses. The normalized confusion matrix further highlights per class correctness: Barrel $\approx$ 0.85, Barrier $\approx$ 0.73, Pothole $\approx$ 0.71, and Pylon $\approx$ 0.80. However, Debris ($\approx$ 0.22) and Log ($\approx$ 0.49) remain weaker, and there is still substantial confusion with the background class. In particular, background to hazard false positives remain large for Debris and Pothole, and many true hazards are still predicted as background (false negatives). This suggests Model 2 is much improved overall, but performance is still constrained by difficult classes (Debris) and background look alikes.

4.3.4 Model 1 vs Model 2 Comparison

Model 2 improves substantially over Model 1 due to (1) a larger labeled dataset and (2) revised class definitions that reduce class visual variation (splitting Construction into Barrel/Barrier/Pylon and removing Snow). Quantitatively, overall mAP 50 increases from 0.412 (Model 1) to 0.565 (Model 2). Overall, Model 2 provides a significantly more stable and accurate detector for the hazards most relevant to the navigation and safety scoring components of the system.

5 Intelligent Navigation System

The Intelligent Navigation System integrates graph based pathfinding, computer vision derived safety metrics, and a web based user interface to provide route recommendations that account for both efficiency and road safety. The system models an urban road network as a weighted graph, where intersections are represented as nodes and road segments are represented as edges. Each edge may be associated with a safety score derived from hazard detection, allowing routing decisions to extend beyond traditional shortest distance navigation.

5.1 Pathfinding Algorithms

This application uses two pathfinding algorithms to achieve different routing objectives. Dijkstra's algorithm is used to compute the shortest path between a selected start and destination node when prioritizing distance or travel time (without considering safety). This provides a baseline "fastest route". For safety aware routing, A* search is used with a heuristic that incorporates both spatial distance and edge safety weights. This heuristic guided approach enables efficient exploration of the graph while favoring road segments with lower hazard risk, producing a "safest route". In addition, a combined routing mode balances distance and safety by adjusting edge weights to reflect both factors.

5.2 User Interface

The application can be accessed through a web based user interface built using Flask and rendered in the user's browser. Start and destination locations are selected via dropdown menus that are dynamically populated from the graph's node list, ensuring that users can only choose valid intersections. The interface provides three routing options through dedicated buttons: fastest route, safest route, and combined route. Route results are displayed on an interactive map, allowing users to compare paths. The interface also allows users to upload an image and associate it with a specific road (graph edge), enabling dynamic updates to the safety evaluation of that segment.

5.3 Safety Simulation and Hazard Evaluation

To demonstrate system functionality without requiring live data collection, hazard detection and safety evaluation are simulated. Each road segment may have one or more images manually associated with it. When generating a safety aware route, the system locally processes these images using the trained object detection model to identify hazards and compute a normalized safety score for the corresponding edge. These safety scores are then used as weights in the routing algorithms, allowing the system to dynamically adjust routes based on detected road conditions.

5.4 System Architecture and Workflow

The application is launched by running app.py, which starts the Flask server. The application can be accessed at a local web address that prints to terminal. Once the user accesses the interface, they select a start and end intersection. The system computes the fastest route based solely on graph distance. It then checks a JSON file to determine whether any images are associated with the edges along the route. If images are present, the system evaluates them for hazards using the trained model, updates the safety scores of the relevant edges, and recalculates the route to produce a safer alternative. The user can choose between the shortest route, the safest route, or a combined route that balances speed and safety, with all routes visualized on the map.

6 Challenges Faced

We faced several challenges while completing this project. The first issue was that we found it difficult to find enough road hazards to effectively train our model. We used Google Studio to augment images, introducing artificial hazards and variations in lighting, weather, and visibility conditions. Our hope was that this would help to improve model generalization and simulate realistic urban scenarios that may not be consistently present in the original dataset.

Despite adding hazards to improve our detection model, the trained model did not identify hazards effectively. We initially trained the model on 369 annotated images, including the following hazard categories: Construction, Logs, Debris, Snow, and Potholes.

In hopes of increasing the effectiveness of our model training, we decided to change the hazard categories in hopes of achieving better detection accuracy. We split the Construction category into the more specific categories of Pylon, Barrier, Barrel. We also eliminated snow as a category. Furthermore, we annotated a fresh set of 1125 images.

As a result of the changes, we found that the model's hazard detection accuracy was improved.

7 Conclusions

This project successfully demonstrates a proof of concept system that integrates computer vision, graph based navigation, and geospatial visualization to support safety aware routing. By training and fine tuning a YOLOv11x model, we were able to detect road hazards from images and translate them into weighted safety scores that influence route selection. The Flask based backend, JSON storage, and Leaflet/Folium interface allowed us to create a modular, interactive prototype that highlights how detected hazards can inform navigation decisions beyond traditional shortest path calculations.

Through this work, we gained practical experience with image annotation, convolutional neural network training, and geospatial visualization. The iterative improvements to model training and hazard classification highlighted the importance of dataset quality and using precise class definitions. While the system currently operates with manually associated images, its design makes it straightforward to scale to larger datasets or even real time hazard detection. We believe that this project demonstrates the potential for combining computer vision and spatial analytics to enhance road safety.

References

[1] Mapillary, "Mapillary," *Mapillary*. [Online]. Available: `https://www.mapillary.com/`. Accessed: Nov. 15, 2025.

[2] Google, "Nano Banana - Gemini AI image generator & photo editor," *Gemini*. [Online]. Available: `https://gemini.google/overview/image-generation/`. Accessed: Nov 16. 2025.

[3] E. Juras, "How to Train YOLO 11 Object Detection Models Locally with NVIDIA," *EJ Technology Consultants*, Dec. 30, 2024. [Online]. Available: `https://www.ejtech.io/learn/train-yolo-models`. Accessed: Nov. 24, 2025.

[4] Pallets Projects, "Welcome to Flask — Flask Documentation (3.1.x)," *Flask Documentation*. [Online]. Available: `https://flask.palletsprojects.com/en/stable/`. Accessed: Dec. 10, 2025.

[5] R. Story, "Folium — Python data, leaflet.js maps (Folium 0.20.0 documentation)," *Folium Documentation*. [Online]. Available: `https://python-visualization.github.io/folium/latest/`. Accessed: Dec. 10, 2025.

[6] Ultralytics, "Ultralytics YOLO11," *Ultralytics YOLO Docs*. [Online]. Available: `https://docs.ultralytics.com/models/yolo11/`. Accessed: Nov. 18, 2025.