Development and evaluation of a vision-based pavement distress detection and auto-labeling framework for transportation agency pavement monitoring

by

Deekshitha Chowdary Kalluri

B.S., University of South Florida, 2024

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2026

Approved by:

Major Professor

Dr. Lior Shamir

Copyright

© Deekshitha Chowdary Kalluri 2026.

Abstract

State transportation agencies increasingly collect roadway imagery at a scale that makes frame-by-frame manual annotation expensive to sustain. The primary contribution of this thesis is a reproducible pavement-distress auto-labeling and retraining pipeline for agency-style roadway imagery. The system combines YOLO11x detection, ByteTrack temporal association, temporal-persistence filtering, representative-frame selection, GPS-overlay OCR, quality assurance, and retraining. A controlled experiment evaluates whether the candidate-selection stages can reduce roadway training-data volume without materially changing downstream detector performance.

Nine pavement-distress sources were harmonized into a six-class corpus containing 98,661 retained images and 182,308 bounding-box annotations. The final experiment used a fixed public base of 78,928 training images, 9,866 validation images, and 9,867 frozen test images. Four YOLO11x conditions were compared under the same training and evaluation protocol: public-only control; filtered/subsampled training with 1,060 KDOT auto-labeled frames; equal-size random sampling of 1,060 frames from the corrected detector-positive candidate pool; and corrected unfiltered training with 3,266 KDOT detector-positive frames.

On the frozen public test set, the control, filtered, equal-size random, and unfiltered conditions achieved mAP50 values of 0.665, 0.664, 0.661, and 0.667 and mAP50–95 values of 0.383, 0.381, 0.381, and 0.383, respectively. The filtered condition exceeded the same-size random condition by 0.003 mAP50, with equal mAP50–95 to three decimals. The filtered/subsampled condition therefore used 67.5% fewer added KDOT training frames than the unfiltered condition while differing by only 0.003 mAP50 and 0.002 mAP50–95. Because temporal-persistence filtering and representative subsampling changed together, the experiment evaluates the combined candidate-selection strategy rather than the isolated causal effect of temporal persistence. A single training seed was used, so the small metric differences are not presented as statistically significant.

Qualitative review showed both the value and the limitation of temporal selection: an isolated false crack response was absent from the final selected arm, while a false Patch prediction on an adjacent vehicle survived selection. The result is therefore operational rather than an accuracy-improvement claim: the implemented workflow substantially reduced added roadway training-data volume while preserving essentially comparable performance on the common public-domain test set, and retained pseudo-labels still required quality assurance.

Keywords: Pavement distress detection; Auto-labeling; YOLO; Object tracking; Temporal persistence; Transportation infrastructure

Contents

List of Figures vii

List of Tables x

List of Abbreviations xi

1 Introduction 1

1.1 Problem Statement . . . 1
1.2 Project Context . . . 1
1.3 Research Questions and Contributions . . . 2
1.4 Thesis Organization . . . 3
1.5 Pipeline Overview . . . 3

2 Literature Review 5

2.1 From Manual Inspection to Image-Based Pavement Assessment . . . 5
2.2 Crack Segmentation and Fine-Grained Surface Defect Recognition . . . 5
2.3 Object Detection for Road and Pavement Damage . . . 6
2.4 Vehicle-Mounted and Maintenance-Oriented Systems . . . 7
2.5 Public Pavement-Distress Datasets and Domain Heterogeneity . . . 8
2.6 Weak Supervision and Semi-Supervised Object Detection . . . 8
2.7 Temporal Consistency and Multi-Object Tracking . . . 9
2.8 Research Gap and Thesis Positioning . . . 9

3 Dataset 11

3.1 Source Datasets and Local Audit . . . 11
3.2 KDOT Roadway Video Acquisition . . . 14
3.3 Class Taxonomy and Remapping . . . 14
3.4 Dataset Quality Assurance and Negative Samples . . . 15
3.5 Dataset Statistics . . . 16
3.6 Train, Validation, and Test Partition . . . 17

4 Methodology 18

4.1 Baseline Model Training . . . 18
4.2 Evaluation Metrics . . . 19

4.3 Auto-Labeling Pipeline Architecture 20
4.3.1 Motivation: Why Temporal Filtering? 20
4.3.2 Pipeline Stages 21
4.3.3 Implementation Notes 23
4.4 Iterative Retraining Workflow 26
4.5 Experimental Design 27

5 Results **30**
5.1 Final Frozen-Test Aggregate Performance 30
5.2 Per-Class Frozen-Test Performance 31
5.3 Filtered/Subsampled vs. Unfiltered Candidate Volume 32
5.4 Equal-Size Random Selection Control 33
5.5 Final-Test Dataset Behavior 34
5.6 Final Dataset Integrity and Manual-Audit Status 34
5.7 Targeted Lane-Marking Review 35
5.8 Qualitative Auto-Label Failure Analysis 35
5.9 Pipeline Throughput 36

6 Discussion **37**
6.1 Core Finding: Data Reduction with Comparable Public-Test Performance . . 37
6.2 What the Comparison Does and Does Not Isolate 37
6.3 Interpretation of the Small Metric Differences 38
6.4 Domain Generalization and the Meaning of the Frozen Test Set 38
6.5 Failure Modes and the Role of Quality Assurance 38
6.6 Practical Implications for KDOT 39
6.7 Threats to Validity 39
6.8 Limitations 40
6.9 Data and Code Availability 41

7 Conclusion **42**
7.1 Summary 42
7.2 Contributions 42
7.3 Future Work 43
7.4 Closing Remarks 43

References **45**

A Final Evaluation Provenance **48**

B Software Artifact and Reproducibility Guide **49**
B.1 Quick Start — I Just Want to Run It . . . 49
B.2 Minimum Viable Test . . . 50
B.3 What This Project Does . . . 51
B.4 System Requirements . . . 52
Before You Begin . . . 52
B.5 Installing Prerequisites . . . 53
B.5.1 Windows . . . 53
B.5.2 macOS . . . 53
B.6 Getting the Project . . . 54
B.7 Setting Up the Python Environment . . . 54
Upgrading . . . 55
B.8 Project Folder Structure . . . 56
B.9 Downloading Pretrained Model Weights . . . 56
B.10 Running the Auto-Labeling Pipeline . . . 57
B.10.1 Single video . . . 57
B.10.2 Folder of videos . . . 58
B.10.3 Folder of still images . . . 58
B.10.4 Optional flags . . . 59
B.10.5 GPS OCR notes . . . 59
B.11 Understanding the Output . . . 59
B.12 Running on GPU (Optional but Recommended) . . . 62
B.13 Training or Retraining a Model . . . 63
B.14 Iterative Retraining Cycle . . . 64
B.15 Generating Report Figures and Tables . . . 66
B.16 GPU vs. CPU Notes . . . 66
B.17 Troubleshooting . . . 67
B.18 Command Quick Reference . . . 68
B.19 FAQ . . . 69
B.20 Workflow Decision Tree . . . 70

List of Figures

1.1 End-to-end pavement-distress auto-labeling and retraining pipeline. Raw KDOT roadway video is sampled, processed by YOLO11x and ByteTrack, filtered for temporal persistence, and subsampled to representative frames. GPS-overlay OCR is stored as metadata, candidate labels remain subject to manual quality assurance, and accepted KDOT imagery enters the training population only. 4

3.1 Full-corpus class distribution after harmonization: 182,308 bounding-box annotations across 98,661 retained images. Longitudinal Crack is the dominant class, while Patch and Pothole are comparatively scarce. 17

4.1 Development-stage YOLO11 model-scale comparison. Performance improved from nano to medium to extra-large on both mAP50 and mAP50–95, at a substantial increase in model complexity. YOLO11x was selected for the downstream auto-labeling and controlled retraining experiments. 19

4.2 Real consecutive-frame example of an isolated false detector response. The false Longitudinal Crack candidate appears only in the center frame of the displayed sequence and is absent from the final filtered/subsampled training arm. The example illustrates the behavior of the combined candidate-selection pipeline; it does not isolate the causal contribution of persistence filtering from representative-frame selection. 21

4.3 Consecutive detector-positive frames during an overpass-shadow transition. The corrected unfiltered candidate set contains all five displayed frames, while the final filtered/subsampled arm retains frames 42, 44, and 46. The example demonstrates representative reduction within a temporally consistent sequence under challenging illumination. 21

4.4 GPS coordinate extraction from the on-screen dashcam text overlay. (a) A representative frame showing the burned-in GPS text strip at the bottom of the image; the orange dashed box indicates the region cropped for OCR. (b) Processing steps from raw frame to validated GPS output, including the majority-sign correction applied to handle intermittent sign-inversion errors in the OCR output. The specific coordinate values shown (`39.16799, -96.60373`) are real values read from pipeline run `06_GRMN0032.MP4_output/metadata.csv`. 25

4.5 Four-arm experimental design for the final YOLO11x comparison. All conditions share the same 78,928-image public training base and frozen validation (9,866 images) and test (9,867 images) manifests. The filtered and Random1060 conditions each add 1,060 KDOT frames, whereas the unfiltered condition adds 3,266. KDOT imagery is used for training only. All four conditions use the same model architecture and training configuration, with checkpoints selected using validation performance before evaluation on the identical frozen public test set. 29

5.1 Final frozen-public-test comparison of four YOLO11x training conditions: public-only control, filtered/subsampled KDOT augmentation, equal-size random KDOT sampling (Random1060), and corrected unfiltered KDOT augmentation. All models were evaluated on the same 9,867-image public test set. The observed differences do not establish statistically significant superiority. 31

5.2 Per-class mAP50–95 on the identical frozen public test set. No class shows a large collapse after reducing the KDOT augmentation from 3,266 unfiltered detector-positive frames to 1,060 filtered/subsampled frames. 32

5.3 Reduction in added KDOT training-frame volume. The combined temporal-persistence and representative-frame strategy reduces the detector-positive candidate set from 3,266 to 1,060 frames, or 67.5%. This percentage applies to the added KDOT frames, not to the entire public-plus-KDOT training corpus. 33

5.4 Representative manually reviewed auto-label failures. A false Patch prediction is placed on an adjacent vehicle and remains in the final filtered/subsampled arm; two road-paint/lane-marking crack confusions are present in the corrected unfiltered candidates but absent from the final arm. These examples show that candidate selection can remove some erroneous frames but does not guarantee correctness of retained boxes. 35

5.5 False prediction that survives candidate selection. The same Patch box is present in the corrected unfiltered candidate set and the final filtered/subsampled arm. Temporal persistence and representative selection therefore do not provide a box-level correctness guarantee, and retained candidates still require quality assurance. 36

B.1 A real frame from `images_annotated/`, showing two Longitudinal Crack detections with confidence scores drawn on the dashcam frame. GPS overlay is visible at the bottom. 61

B.2 A second real annotated roadway frame from the same Garmin video, showing two YOLO-predicted Longitudinal Crack regions with confidence scores of approximately 46% and 36%. One prediction overlaps a white roadway marking, illustrating why automatically generated labels require visual quality assurance. 61
B.3 A real `metadata.csv` excerpt opened as a spreadsheet. The `latitude` and `longitude` columns are populated where the source video had a readable GPS overlay; rows with no GPS data have blank coordinate fields, which is expected. 62

List of Tables

2.1 Representative related work and its relationship to the present thesis. 10

3.1 Audited source composition of the 98,661-image harmonized corpus. Resolution information describes the retained local files, not necessarily the complete upstream release. 12

3.2 Intermediate 15-class label mapping to the six-class project taxonomy. A value of "Drop" means that the annotation was intentionally omitted from the six-class training labels. 15

3.3 Train/validation/test split of the remapped experimental dataset. 17

4.1 Verified development-stage YOLO11 scale comparison. GFLOPs were reported by Ultralytics at the 896-pixel thesis input size. The best epoch is selected by the recorded validation mAP50–95 value. 18

4.2 Verified auto-labeling and final-training configuration used for reproducibility. 24

5.1 Final four-arm performance on the identical frozen 9,867-image public test set. 30

5.2 Per-class mAP50 on the frozen public test set for all four conditions. Δ_F denotes filtered/subsampled minus control. 31

5.3 Training-data volume and final frozen-test performance for the four controlled conditions. 32

List of Abbreviations

AI	Artificial intelligence
CSV	Comma-separated values
GPS	Global Positioning System
HPC	High-performance computing
IoU	Intersection over Union
KDOT	Kansas Department of Transportation
mAP	Mean average precision
OCR	Optical character recognition
SLURM	Simple Linux Utility for Resource Management
SSOD	Semi-supervised object detection
UAV	Unmanned aerial vehicle
YOLO	You Only Look Once

Chapter 1 – Introduction

1.1 Problem Statement

State transportation agencies are responsible for monitoring the condition of thousands of lane-miles of pavement on an ongoing basis. Manual inspection—the traditional approach—requires trained crews to drive designated routes, record distress observations by hand, and produce condition estimates that vary between inspectors and between inspection cycles. The process is expensive, slow, and difficult to scale to the frequency that proactive maintenance planning requires.

Deep-learning-based object detection offers a path toward automating distress identification from imagery collected by routine inspection vehicles—whether dashcam video or still images captured by mounted cameras. Trained detectors applied to pavement imagery can, in principle, identify and locate distress automatically, removing the need for a dedicated inspection crew. The practical obstacle is not the detection algorithm itself but the data it requires: a detector performs only as well as the labeled training data used to build it, and manually annotating dashcam-scale video frame by frame is itself a significant labor investment.

This thesis addresses that bottleneck directly. Rather than treating the training dataset as a fixed, one-time investment, this work develops and evaluates a framework that uses an already-trained detector to annotate new, unlabeled pavement imagery automatically, with temporal-persistence filtering applied when sequential video frames are available, and feeds the cleaned result back into model retraining. The central question is whether this self-expanding approach can reduce the recurring annotation burden while preserving controlled downstream detection performance.

1.2 Project Context

This thesis was conducted within Kansas Department of Transportation (KDOT) K-TRAN research project KSU-26-4, KDOT project RE-0922-01, *Use of Advanced Data Capture Tools on Measurements of Crack Lengths and Potholes for Estimates and Final Quantities* [1, 2]. The signed project proposal defines the broader objective as automating the capture and analysis of high-resolution roadway imagery at highway speed, including dataset construction, image-processing algorithm development, and eventual quantity estimation; it also identifies delivery of the developed software as a project deliverable [2]. The public project record motivates this work by noting that conventional measurement practices can require substantial

labor and time, are susceptible to human error, can disrupt traffic, and can place personnel near active traffic [1].

Within that transportation objective, this thesis focuses on a computer-science bottleneck: converting recurring roadway video into useful detector-training data without manually annotating every frame or retaining every detector-positive frame. The developed workflow combines YOLO-based distress detection, ByteTrack temporal association, temporal-persistence filtering, representative-frame selection, OCR-based GPS metadata extraction, manual quality assurance, controlled retraining, and frozen-set evaluation.

The original project scope permitted evaluation of both segmentation-oriented and single-stage detection approaches. This work adopts the YOLO path. Once a practical baseline detector had been established, dataset construction and recurring annotation cost became the dominant engineering and research constraints, motivating the temporal auto-labeling and data-selection work evaluated in this thesis.

1.3 Research Questions and Contributions

Three questions drive this work:

1. Can detector-generated labels selected from temporally consistent dashcam detections be incorporated into an augmented retraining corpus without a large observed degradation on the frozen public-domain test set?

2. How does retraining with the filtered/subsampled candidate set compare with retraining on all detector-positive frames from the same footage, particularly with respect to downstream detection performance and retained training-data volume?

3. What end-to-end processing throughput was observed for the implemented auto-labeling pipeline under the available GPU computing environment?

This work makes the following contributions:

1. A reproducible roadway-video auto-labeling framework that accepts video or still-image input and combines YOLO11x detection, ByteTrack temporal association, temporal-persistence filtering, representative first/middle/last sampling, GPS-overlay OCR, human-review artifacts, and YOLO-format training output (Chapter 4).

2. An audited six-class pavement-distress corpus assembled from nine locally retained source groups, including exact retained image counts, observed local image formats/resolutions, a reproducible 15-to-6 class-remapping procedure, and explicit documentation of source groups whose upstream provenance could not be reconstructed reliably (Chapter 3).

3. A controlled four-arm experiment comparing public-only training, 1,060 filtered/subsampled KDOT frames, an equal-size random sample of 1,060 corrected KDOT frames, and 3,266 corrected unfiltered KDOT detector-positive frames while holding the public validation and test populations fixed (Sections 4.5 and 5.3).

4. A data-efficiency result showing that the combined temporal-selection strategy reduced added KDOT training-frame volume by 67.5% while preserving comparable frozen-public-test performance; the experiment is not presented as an isolated causal ablation of temporal persistence.

5. A qualitative failure analysis showing both sides of temporal selection: an isolated false crack response that is absent from the final filtered/subsampled arm, temporally repeated detections retained under challenging shadow, and a visually clear false Patch prediction that survives selection. These cases establish that candidate selection reduces data volume but is not a substitute for label-quality assurance.

6. A reproducibility and handoff record covering the final frozen-test manifest, training/evaluation configuration, software versions, SLURM jobs, checkpoint hashes, and one authoritative user and operations manual shared by the thesis appendix, GitHub repository, and KDOT handoff.

1.4 Thesis Organization

Chapter 2 surveys related work in vehicle-mounted detection, annotation cost reduction, multi-object tracking, and GPS integration. Chapter 3 describes the construction of the training dataset. Chapter 4 presents the auto-labeling pipeline, the retraining workflow, and the experimental design. Chapter 5 reports the results. Chapter 6 interprets those results and discusses limitations. Chapter 7 concludes and identifies directions for future work.

1.5 Pipeline Overview

Figure 1.1 summarizes the end-to-end workflow. The architecture separates frame-level detection from video-level candidate selection, then keeps KDOT imagery confined to training while the original public validation and test manifests remain frozen.

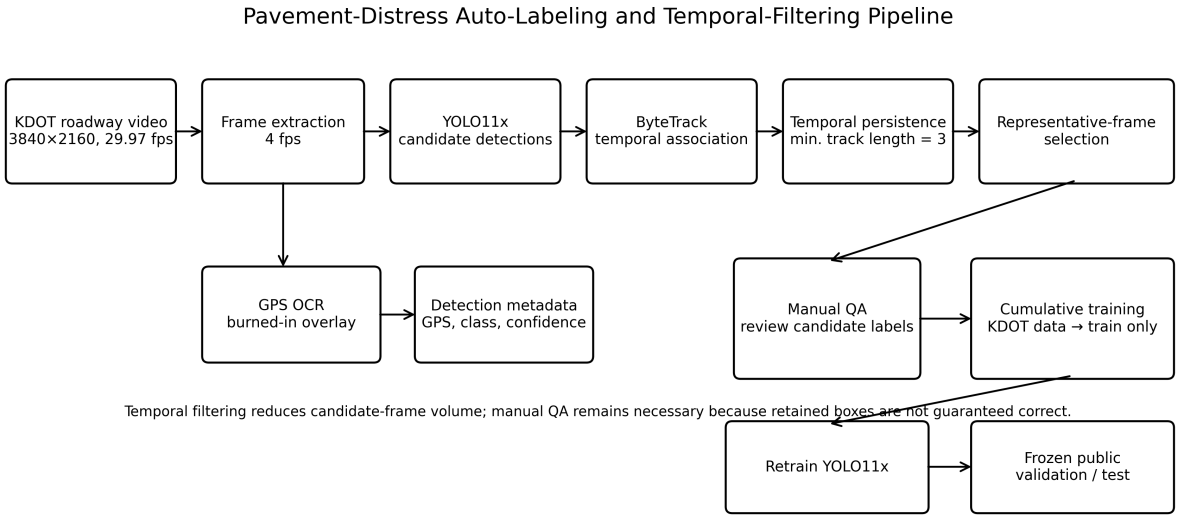


Figure 1.1: End-to-end pavement-distress auto-labeling and retraining pipeline. Raw KDOT roadway video is sampled, processed by YOLO11x and ByteTrack, filtered for temporal persistence, and subsampled to representative frames. GPS-overlay OCR is stored as metadata, candidate labels remain subject to manual quality assurance, and accepted KDOT imagery enters the training population only.

Chapter 2 – Literature Review

Automated pavement-condition assessment spans several related research problems rather than a single detection task. Earlier work focused on handcrafted image-processing methods and crack segmentation; later work increasingly adopted convolutional neural networks, object detectors, mobile sensing, and vehicle-mounted acquisition. More recent research also addresses dataset diversity, limited annotation budgets, and the use of unlabeled data through weakly supervised or semi-supervised learning. Because the present thesis combines a vehicle-mounted detector with detector-generated labels and temporal filtering, the literature review is organized around the research components that directly affect that design.

2.1 From Manual Inspection to Image-Based Pavement Assessment

Manual pavement surveys are expensive, labor intensive, and subject to differences among inspectors. Modern automated systems therefore combine image acquisition with algorithmic recognition of cracks, potholes, patches, and related surface defects. Recent reviews show that the field now includes traditional image processing, classical machine learning, semantic segmentation, object detection, three-dimensional sensing, vehicle-mounted cameras, and mobile or edge inference [16, 17]. The same reviews also identify data quality, dataset variability, generalization, and deployment constraints as persistent problems rather than solved implementation details.

Two task formulations are especially common. Crack-segmentation methods attempt to recover the pixel-level shape of a crack, which is useful for detailed geometric measurement. Object detectors instead return class-labelled bounding boxes and are usually less expensive to annotate and faster to integrate into real-time monitoring systems. This thesis uses the second formulation because its immediate objective is scalable detection and automatic label generation rather than pixel-accurate crack geometry.

2.2 Crack Segmentation and Fine-Grained Surface Defect Recognition

DeepCrack is an influential example of deep crack segmentation. Zou et al. [4] combined hierarchical convolutional features with deep supervision and introduced a benchmark containing 537 manually annotated RGB images. The work illustrates both the advantage and the cost

of segmentation: the output captures crack morphology more precisely than a bounding box, but the required pixel-level annotations are considerably more expensive to produce.

Subsequent surveys of deep-learning crack segmentation report continued improvements from feature fusion, larger receptive fields, attention mechanisms, and hybrid architectures, while also identifying limited dataset size and annotation cost as barriers to generalization [18]. Transformer-based approaches have also been introduced for crack detection. Xiao et al. [21] used hybrid-window attentive vision transformers to model long-range crack structure, while a 2023 transformer-based study combined attention with transformer feature extraction for long and visually complex pavement cracks [22]. These studies illustrate a broader movement from purely local convolutional features toward architectures that can model extended crack geometry.

These observations motivate the use of bounding-box detection in this thesis and also reinforce the broader motivation for reducing the amount of new manual annotation required when additional roadway imagery becomes available.

2.3 Object Detection for Road and Pavement Damage

Maeda et al. [3] demonstrated one of the most influential low-cost road-damage detection systems using smartphone imagery collected from a vehicle. Their dataset contained 9,053 road images and 15,435 annotated damage instances, and the study compared convolutional object-detection methods for classifying multiple damage types. The work helped establish a practical research direction in which ordinary cameras and mobile hardware can support roadway condition assessment without specialized laser equipment.

The Road Damage Dataset family subsequently expanded this line of work across countries and acquisition platforms. RDD2022 contains 47,420 images from Japan, India, the Czech Republic, Norway, the United States, and China, with 55,007 released training annotations across longitudinal cracks, transverse cracks, alligator cracks, and potholes [5]. This multi-national structure is important because pavement appearance varies with climate, pavement material, camera geometry, road markings, and acquisition hardware.

Pothole-specific work demonstrates the continuing tradeoff between accuracy and deployment speed. Asad et al. [10] compared several YOLO variants and SSD-MobileNetV2 on edge-oriented hardware; YOLOv5 achieved the highest reported image-set mAP in that study, while Tiny-YOLOv4 produced the strongest real-time throughput on the tested OAK-D configuration. For transportation agencies, these studies show that detector choice is partly an operational decision: a model that is slightly more accurate but substantially slower may not be preferable when continuous footage must be processed.

Several pavement studies have modified YOLO rather than treating it as a fixed detector.

Guo and Zhang [19] proposed an improved YOLOv5 road-damage detector using MobileNetV3, coordinate attention, revised anchor clustering, label smoothing, and re-parameterization. In a separate multi-distress study, Guo et al. [20] optimized YOLOv5s with attention and deployed it on an intelligent mobile platform under varied road-imaging conditions. More recent work has continued this direction with YOLOv8: Jiang [23] reported an embedded-oriented YOLOv8 variant evaluated on RDD2022, while Wang et al. [24] combined an enhanced YOLOv8 detector with synthetic pavement-distress imagery to address data scarcity. These studies are useful comparators because they improve the detector architecture or training data itself, whereas the present thesis focuses on how new field imagery is selected, labeled, and fed back into retraining.

YOLO11 is used in the present thesis because it provides multiple model scales under a common training and inference framework. Khanam and Hussain [27] describe architectural changes in YOLO11, including C3k2, SPPF, and attention-oriented components. Rather than proposing another YOLO architecture modification, the present work treats the detector as one component inside a larger data-generation and retraining workflow.

2.4 Vehicle-Mounted and Maintenance-Oriented Systems

A detector is more useful to a road agency when it is integrated with a repeatable acquisition and reporting process. PotholeEye+ [9] is an example of a vehicle-mounted system combining a camera, computing hardware, and GPS. It was evaluated during highway driving and reported average accuracy of approximately 92 percent, precision of 87 percent, and recall of 74 percent at an average driving speed of 110 km/h. The study is relevant to the present work not because the two systems use identical models, but because both treat detection as part of an operational vehicle workflow rather than as an offline image-classification benchmark.

Recent reviews of automated pavement data collection similarly emphasize that camera systems, GPS, distance measurement, and AI-based image processing need to be considered together for maintenance applications [16, 25]. End-to-end pavement systems have also begun to connect detection with condition assessment rather than stopping at bounding boxes. For example, a 2024 system combined a YOLOv5 detector, rule-based false-positive filtering, geotagged imagery, and a pavement-condition index for identifying road sections requiring maintenance [26]. The present pipeline follows the same maintenance-oriented systems perspective while concentrating on a different bottleneck: generating and curating additional labeled data from routine dashcam footage.

2.5 Public Pavement-Distress Datasets and Domain Heterogeneity

Dataset heterogeneity is central to pavement-distress research. SVRDD contains 8,000 1024-by-1024 Beijing street-view images with 20,804 bounding-box annotations across multiple pavement-damage categories [6]. HighRPD uses a very different viewpoint: UAV images originally captured at high resolution were cropped into 11,696 640-by-640 images with YOLO-format annotations for line cracks, block cracks, and potholes [7]. Attain contains 2,293 images extracted from smartphone video in Tehran and provides 19,761 distress instances across a broader set of distress categories and severity levels [8].

These datasets differ not only in class names but also in camera geometry, geography, image resolution, annotation format, and distress granularity. A simple concatenation of such datasets would therefore create inconsistent supervision. The class-remapping procedure described in Chapter 3 is a necessary modelling step: it defines which source categories are treated as equivalent in the six-class taxonomy used for this project.

2.6 Weak Supervision and Semi-Supervised Object Detection

The annotation bottleneck addressed in this thesis is closely related to weakly supervised and semi-supervised learning. In pavement analysis, WSPLIN reduces annotation demand by inferring patch labels from image-level supervision and exploiting the assumption that distress occupies a relatively sparse part of a high-resolution pavement image [11]. The approach is spatial: it uses structure within an image to infer more localized information from weaker labels.

Semi-supervised object detection (SSOD) addresses a related problem in a broader object-detection setting. Instant-Teaching [12], Unbiased Teacher [13], and Soft Teacher [14] all use model-generated pseudo-labels on unlabeled data while attempting to reduce the effect of unreliable pseudo boxes or confirmation bias. These methods are important context for the present work because they show that detector-generated labels can be useful, but also that pseudo-label quality cannot be assumed. A model can reinforce its own mistakes if inaccurate predictions are repeatedly treated as ground truth.

The present pipeline differs from these end-to-end SSOD methods. It does not jointly optimize a teacher and student during one training process. Instead, it is an engineering-oriented offline loop: a trained detector labels new roadway imagery, a temporal rule is applied to video-derived candidates, the retained labels are merged into a new training corpus,

and a new detector is trained. The temporal filtering stage is therefore best understood as a domain-specific pseudo-label selection rule rather than as a replacement for the broader SSOD literature.

Terminology is used deliberately in the remainder of this thesis. *Auto-labeling* describes the implemented software workflow; *pseudo-labels* refers to labels predicted by a model rather than supplied by a human annotator; and *weak supervision* is reserved for methods whose supervision signal is intrinsically coarser or indirect, such as image-level labels used to infer patch-level information. These concepts overlap in their goal of reducing annotation cost but are not treated as interchangeable.

2.7 Temporal Consistency and Multi-Object Tracking

Video provides information that is absent from an unordered image collection. A real pavement feature remains visible across a sequence of frames as the camera approaches and passes it, whereas many transient detector responses do not persist in the same way. Exploiting this signal requires associating detections over time.

ByteTrack [15] associates detection boxes with tracklets and, unlike strategies that immediately discard lower-confidence detections, attempts to recover true objects from lower-score boxes when they remain consistent with an existing trajectory. The method was originally evaluated on conventional multi-object-tracking benchmarks, not on pavement features observed from a moving camera. The present thesis therefore uses ByteTrack as a practical association mechanism but does not assume that its benchmark performance directly transfers to the pavement domain. Tracker-ID reuse, viewpoint change, frame sampling rate, and vehicle speed all become relevant implementation concerns, as discussed in Chapter 4.

2.8 Research Gap and Thesis Positioning

The reviewed literature establishes several mature components: deep detectors can recognize pavement damage; vehicle-mounted systems can operate at road speed; public datasets support increasingly diverse training; and pseudo-labeling can reduce reliance on fully labeled data. However, fewer studies evaluate the specific operational loop considered here: using a deployed pavement detector to generate new training labels from agency-style dashcam footage, exploiting video temporal information to reduce candidate redundancy, retraining on the resulting data, and measuring whether the augmented model remains stable on a held-out test set.

Table 2.1 summarizes representative work by research objective rather than presenting a binary feature checklist.

The contribution of this thesis is therefore not a claim that temporal persistence is

Table 2.1: Representative related work and its relationship to the present thesis.

Study	**Input / data**	**Primary task**	**Relevant contribution**	**Limitation relative to this thesis**
Maeda et al. [3]	Vehicle smartphone images	Object detection	Low-cost multi-class road-damage detection	Fixed labeled dataset; no video-based relabel/retraining loop
PotholeEye+ [9]	Highway video + GPS	Detection / classification	Vehicle-mounted operational system	No iterative pseudo-label training evaluation
WSPLIN [11]	Pavement images	Weakly supervised recognition	Reduces annotation burden using spatial patch inference	Does not exploit temporal persistence in roadway video
Unbiased/Soft Teacher [13, 14]	Labeled + unlabeled images	Semi-supervised object detection	Formal pseudo-label learning with teacher–student models	General object-detection setting rather than agency dashcam workflow
ByteTrack [15]	Video detections	Multi-object tracking	Associates detections across time	Designed for conventional MOT benchmarks, not stationary pavement features under camera motion

universally superior to ordinary confidence filtering. Instead, the work evaluates a complete, reproducible pavement auto-labeling workflow and asks an operational question that frame-level detector benchmarks do not answer: how much detector-positive video must be retained for retraining before additional highly correlated frames cease to provide a measurable downstream benefit under a fixed evaluation? This framing makes a near-zero performance change informative rather than inconclusive: if a much smaller selected set and a much larger detector-positive set produce essentially the same held-out result, the additional video-derived frames have low marginal value under that evaluation. The controlled experiment therefore quantifies retained training volume, downstream detector performance, selected false-positive behavior, and processing throughput while keeping its causal claims deliberately narrow.

Chapter 3 – Dataset

The training corpus used in this work was assembled from nine source collections that were not originally designed to share one taxonomy, image geometry, or annotation format. This chapter distinguishes two kinds of information that are easy to conflate: (1) the characteristics reported by the original dataset publishers, and (2) the characteristics of the exact local copies that were retained after preprocessing and remapping for this thesis. The first can be documented from public sources; the second must be audited directly from the project files.

3.1 Source Datasets and Local Audit

The public corpus was assembled from nine locally retained source groups. Because the files were merged over the course of the project, local folder names do not always preserve enough information to reconstruct a canonical upstream dataset release. This thesis therefore reports public provenance only where it can be tied defensibly to the retained group and separately reports the exact characteristics audited from the local corpus. The complete curated project corpus is also archived at `https://huggingface.co/datasets/Deeksha9/pavement-distress-detection`.

Table 3.1: Audited source composition of the 98,661-image harmonized corpus. Resolution information describes the retained local files, not necessarily the complete upstream release.

Local source group	Retained images	Local file type	Observed retained resolutions	Provenance / interpretation
SVRDD	8,000	8,000 JPG	1024×1024 only	Matches the published SVRDD image count and geometry [6].
`archive`	8,132	8,132 JPG	418 distinct sizes; 640×640 is dominant (6,643 images), with 838×471, 1059×596, 1920×1080, 4032×3024, and other sizes also present	Legacy project-import group. The exact canonical upstream identity was not re-established from the retained local files, so no upstream dataset is asserted here.
RDD2022	38,385	38,385 JPG	10 sizes; most common are 600×600 (13,016), 720×720 (7,706), 640×640 (4,805), 512×512 (4,378), 4040×2035 (4,342), and 3643×2041 (2,896)	Retained subset of RDD2022 [5]; filenames preserve country/acquisition subgroups including China, Czech Republic, and India.
HighRPD	11,696	11,696 JPG	640×640 only	Matches the published HighRPD tile count/geometry [7].
`Potholes`	665	665 JPG	117 distinct sizes; 720×720 is most common (285), followed by several 300–533 pixel geometries	Legacy public-source import retained under the local `Potholes` prefix. The folder name alone is not treated as sufficient evidence for a canonical upstream release.

Local source group	Retained images	Local file type	Observed retained resolutions	Provenance / interpretation
`Uncracked`	16,000	15,000 JPG; 1,000 PNG	22 sizes; dominated by 227×170 (10,925), 170×227 (4,012), and 1440×1442 (1,000)	Legacy operational source group. Upstream provenance could not be reconstructed reliably from the retained project files.
`RDO_CVM/RDO`	6,093	6,093 JPG	640×640 only	Legacy operational source group. Upstream provenance could not be reconstructed reliably from the retained project files.
Attain	637	637 JPG	640×640 only	Retained Attain source group; public Attain release described by the dataset authors [8].
RoadDamage Dataset	9,053	9,053 JPG	600×600 only	Retained road-damage imagery with location-coded filenames (e.g., Adachi); corresponds to the vehicle-mounted smartphone road-damage work of Maeda et al. [3].
Total	**98,661**			

The audit shows substantial heterogeneity before any class remapping: some source groups are fixed square tiles, whereas others contain hundreds of distinct geometries and mixed acquisition conditions. For reproducibility, this thesis therefore distinguishes the *local retained corpus* from the complete published size of any upstream dataset. No unresolved local prefix is assigned a public identity or URL by inference alone.

3.2 KDOT Roadway Video Acquisition

The project archive contains three KDOT roadway-video collections totaling 375 MP4 files: 26 files in `vid1`, 162 in `vid2`, and 187 in `vid3`. These counts describe the archived footage, not the amount used in the final controlled training comparison. The experiment in this thesis derives its KDOT augmentation frames from `vid1`; `vid2` and `vid3` remain additional project footage and are not represented as training data in the final four-arm experiment.

All 26 `vid1` clips were audited directly. They share a 3840×2160-pixel (4K) HEVC format at approximately 29.97 frames/s. Together they contain 46,553 raw video frames and approximately 1,553.3 seconds (25.89 minutes) of footage. The source clips are therefore much denser temporally than the data presented to the detector: the processing pipeline samples input video at four frames/s before detection/tracking.

The roadway footage contains burned-in on-screen location information. The pipeline reads this overlay using OCR and stores latitude, longitude, OCR confidence, and sign-correction metadata separately from the five-field YOLO training annotations. Because these coordinates are recovered from rendered text and can be affected by blur, compression, or OCR errors, they are treated as approximate route context rather than survey-grade positioning. Similarly, physical crack length is not inferred directly from bounding-box size because a defensible metric conversion would require camera calibration, including camera geometry and mounting parameters.

This acquisition context is important to the thesis motivation. Recurring high-frame-rate roadway video produces many visually redundant frames, so the data-management problem is not only whether a detector can find distress, but how to select a manageable and useful subset of detector-positive frames for review and retraining.

3.3 Class Taxonomy and Remapping

The source datasets use inconsistent names and levels of granularity for visually related pavement defects. The unified target used throughout the experiments contains six classes: Longitudinal Crack, Transverse Crack, Alligator Crack, Pothole, Patch, and Block Crack. The remapping was implemented explicitly in `scripts/remap_labels.py`; Table 3.2 reproduces that mapping rather than reconstructing it from memory.

Table 3.2: Intermediate 15-class label mapping to the six-class project taxonomy. A value of "Drop" means that the annotation was intentionally omitted from the six-class training labels.

Old ID	Intermediate class	New ID	Project class
0	Longitudinal Crack	0	Longitudinal Crack
1	Transverse Crack	1	Transverse Crack
2	Alligator Crack	2	Alligator Crack
3	Pothole	3	Pothole
4	Manhole Cover	–	Drop
5	Longitudinal Patch	4	Patch
6	Transverse Patch	4	Patch
7	Block Crack	5	Block Crack
8	Sealed Longitudinal	0	Longitudinal Crack
9	Sealed Transverse	1	Transverse Crack
10	Line Crack	0	Longitudinal Crack
11	Oblique Crack	1	Transverse Crack
12	Repair	4	Patch
13	Lateral Crack	0	Longitudinal Crack
14	Other Corruption	–	Drop

Thus, four intermediate labels map to Longitudinal Crack (IDs 0, 8, 10, and 13), three map to Transverse Crack (IDs 1, 9, and 11), three map to Patch (IDs 5, 6, and 12), while Alligator Crack, Pothole, and Block Crack retain one-to-one mappings. Manhole Cover and Other Corruption are excluded from the six-class target. Bounding-box geometry is retained in YOLO normalized center-x, center-y, width, and height form after the class identifier is remapped.

This table describes the merged intermediate taxonomy used by the project after source ingestion; it should not be interpreted as claiming that every upstream source originally used all 15 labels.

3.4 Dataset Quality Assurance and Negative Samples

Later in the project, dataset quality assurance was formalized to distinguish unusable data from legitimate negative examples. A clear roadway image is not considered invalid merely because it has no distress annotation. If an image is visually usable and contains no target distress, an empty YOLO label file represents a valid negative example and is retained. Images are excluded only when the image itself is unsuitable for training or review—for example, when it is too dark, too unclear, corrupted, or otherwise unusable.

The QA workflow checks at least the following conditions:

1. every retained image has a matching label file and every label file has a matching image;

2. each non-empty YOLO label row contains the expected five fields;
3. class identifiers fall within the six-class taxonomy;
4. normalized bounding-box values are valid and non-negative;
5. duplicate annotation lines and duplicate filenames are identified;
6. empty label files are retained when they correspond to valid negative images rather than treated automatically as missing annotation;
7. visually unusable images are removed independently of whether annotations are present.

This QA policy is documented because it affects how the final project corpus should be maintained. It should not be read as a retroactive claim that every later QA action was applied before the experiments in Chapters 4 and 5. The reported model metrics correspond to the dataset state used at training time. Any post-experiment cleanup must therefore be recorded separately rather than silently changing the description of the experimental corpus.

3.5 Dataset Statistics

The remapped experimental corpus contains 98,661 retained images and 182,308 bounding-box annotations across the six target classes. The audited full-corpus counts are 81,660 Longitudinal Crack boxes (44.79%), 40,922 Transverse Crack boxes (22.45%), 20,688 Alligator Crack boxes (11.35%), 11,587 Pothole boxes (6.36%), 9,944 Patch boxes (5.45%), and 17,507 Block Crack boxes (9.60%). Figure 3.1 shows the resulting imbalance.

The frozen test manifest contains 9,867 images. In the final Ultralytics detection evaluation, 18,137 bounding-box instances were used: 8,004 Longitudinal Crack, 4,163 Transverse Crack, 2,112 Alligator Crack, 1,108 Pothole, 916 Patch, and 1,834 Block Crack instances. The evaluator also reported 85 segment records in the mixed-format source data; because the number of segments did not match the number of boxes, segment geometry was discarded and the final comparison used boxes only. This behavior was identical for all four final training conditions.

A filesystem audit found 98,662 label files because one zero-byte orphan label remained in the remapped label directory, whereas 98,661 retained image stems were present and no retained image was missing a matching label. The reported corpus size therefore uses the 98,661 retained images and their valid associated annotations rather than counting the orphan file as an additional sample.

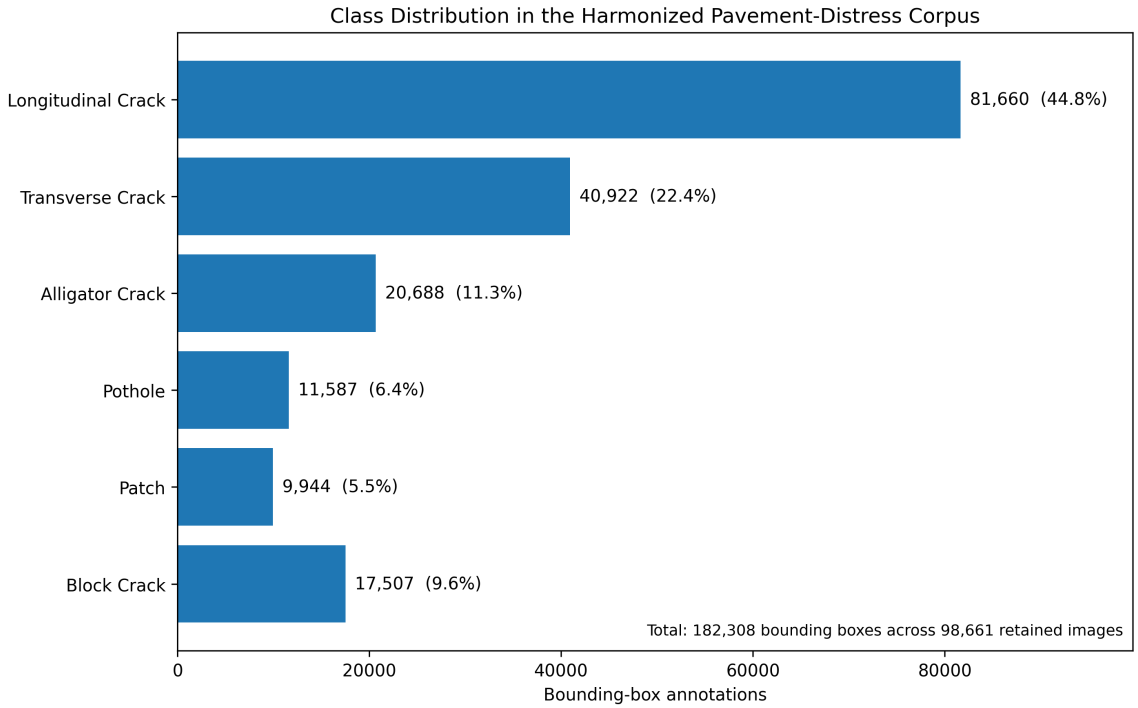


Figure 3.1: Full-corpus class distribution after harmonization: 182,308 bounding-box annotations across 98,661 retained images. Longitudinal Crack is the dominant class, while Patch and Pothole are comparatively scarce.

3.6 Train, Validation, and Test Partition

The reported baseline split used an 80/10/10 partition with random seed 42. Table 3.3 reports the image counts used in the original experiment.

Table 3.3: Train/validation/test split of the remapped experimental dataset.

Subset	**Image Count**	**Approximate %**
Training	78,928	80.0%
Validation	9,866	10.0%
Test	9,867	10.0%
Total	**98,661**	**100%**

For the final controlled experiment, the original public validation and test identities were held fixed across all four training conditions. The 78,928-image public training manifest formed the common training base. The 9,866-image validation manifest was used for model selection, and the 9,867-image test manifest remained frozen until final evaluation. The KDOT auto-labeled images were added only to the training manifests; no KDOT image entered the validation or test partitions.

Chapter 4 – Methodology

Building on the dataset described in Chapter 3, this chapter presents the baseline training configuration, the evaluation metrics used for comparison, the auto-labeling pipeline architecture, the iterative retraining workflow, and the experimental design.

4.1 Baseline Model Training

During model development, three YOLO11 scales—nano (n), medium (m), and extra-large (x)—were trained on the remapped public corpus under the recorded project configuration [27]. The development runs used 896-pixel input images, seed 42, cosine learning-rate scheduling, mosaic 1.0, mixup 0.1, scale augmentation 0.8, perspective 0.001, and up to 100 epochs. YOLO11n and YOLO11m used batch size 16, whereas YOLO11x used batch size 8. The archived `args.yaml` records for the medium and extra-large runs point to `last.pt`, consistent with resumed development runs; the comparison is therefore reported as the recorded development model-scale study rather than as three guaranteed-identical from-scratch launches.

Table 4.1: Verified development-stage YOLO11 scale comparison. GFLOPs were reported by Ultralytics at the 896-pixel thesis input size. The best epoch is selected by the recorded validation mAP50–95 value.

Model	Params (M)	GFLOPs	Ckpt (MB)	Epoch	P	R	mAP50	mAP50–95
YOLO11n	2.59	12.6	5.25	100	0.625	0.521	0.568	0.302
YOLO11m	20.06	133.7	38.68	95	0.669	0.594	0.645	0.364
YOLO11x	56.88	383.1	109.13	95	0.676	0.600	0.651	0.375

Model selection. YOLO11x was selected for the downstream auto-labeling and controlled retraining experiments because it achieved the strongest recorded development performance among the three tested YOLO11 scales. YOLO11n achieved 0.568 mAP50 and 0.302 mAP50–95, YOLO11m improved these values to 0.645 and 0.364, and YOLO11x achieved 0.651 and 0.375, respectively. The improvement from YOLO11m to YOLO11x was modest relative to the increase in computational cost: YOLO11x contains 56.88 million parameters and requires 383.1 GFLOPs at the thesis input size, compared with 20.06 million parameters and 133.7 GFLOPs for YOLO11m. Nevertheless, the workflow in this thesis performs offline auto-label generation and retraining on GPU-equipped computing infrastructure rather than resource-constrained real-time edge inference. Because pseudo-label quality was prioritized over model compactness for this stage of the project, the highest-performing development model,

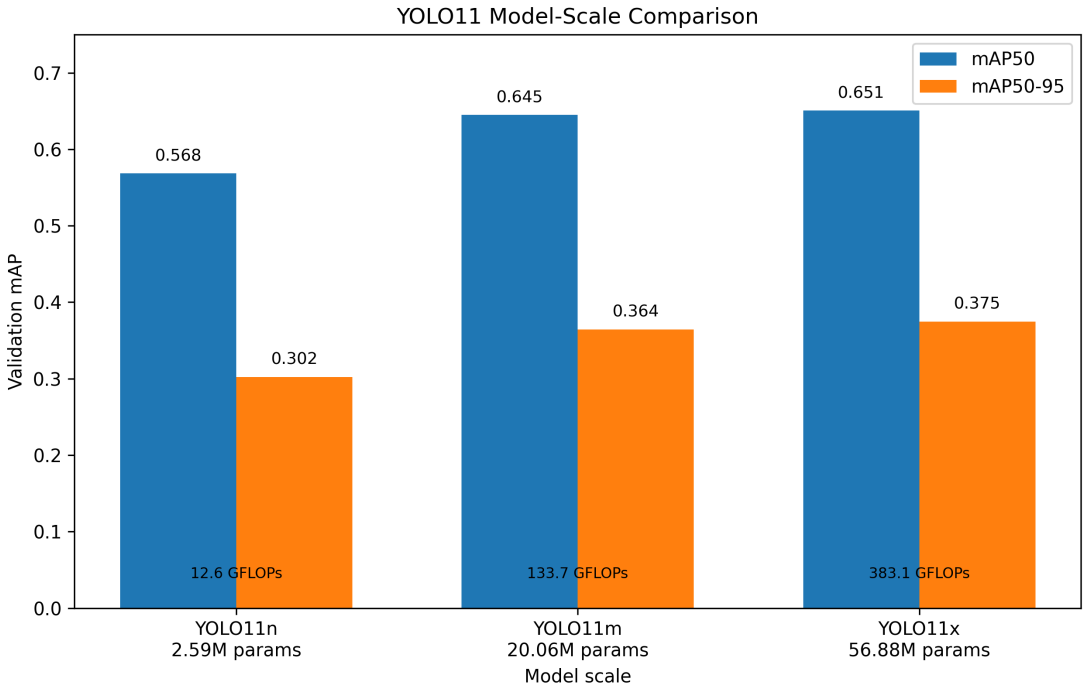


Figure 4.1: Development-stage YOLO11 model-scale comparison. Performance improved from nano to medium to extra-large on both mAP50 and mAP50–95, at a substantial increase in model complexity. YOLO11x was selected for the downstream auto-labeling and controlled retraining experiments.

YOLO11x, was selected as the common detector architecture for the subsequent experiments. All four final experimental conditions therefore use YOLO11x so that architecture is held constant while only the training-data composition changes.

These development results are not the final controlled four-arm experiment. In the final comparison, all four conditions use YOLO11x, share the same public validation and test manifests, and select checkpoints without using the frozen test set.

4.2 Evaluation Metrics

All reported detector comparisons use standard object-detection metrics. Intersection over Union (IoU) measures the overlap between a predicted bounding box B_p and a ground-truth box B_g:

$$\mathrm{IoU}(B_p, B_g) = \frac{|B_p \cap B_g|}{|B_p \cup B_g|}. \tag{4.1}$$

A prediction contributes to the true-positive count when it is assigned to the correct class and satisfies the evaluation procedure's IoU matching criterion; unmatched predictions contribute false positives, and unmatched ground-truth instances contribute false negatives.

Precision and recall are defined as

$$P = \frac{TP}{TP + FP}, \qquad R = \frac{TP}{TP + FN}, \tag{4.2}$$

where TP, FP, and FN denote true positives, false positives, and false negatives, respectively. Precision describes the fraction of reported detections that are correct, while recall describes the fraction of ground-truth objects recovered by the detector.

Average precision (AP) summarizes the precision–recall curve for one class. The principal metric used in this thesis, mAP50, is the mean AP across the six distress classes when detections are evaluated at IoU = 0.50. The more stringent mAP50–95 metric averages mAP over IoU thresholds from 0.50 through 0.95 in increments of 0.05. Because AP integrates performance over the precision–recall curve, the single precision and recall values printed by the evaluation tool should not be interpreted as equivalent to mAP. All baseline-versus-augmentation comparisons use metrics produced by the same standalone Ultralytics validation procedure unless explicitly stated otherwise.

4.3 Auto-Labeling Pipeline Architecture

4.3.1 Motivation: Why Temporal Filtering?

A frame-independent auto-labeling strategy retains every detector-positive image above a confidence threshold, even though adjacent video frames are strongly correlated. This creates two problems: repeated views of the same physical feature can dominate the candidate set, and isolated detector errors can be admitted as if they were equally useful training examples. Temporal information provides an additional selection signal because roadway distress and the camera viewpoint evolve continuously across nearby frames.

The thesis does *not* assume that every temporally persistent prediction is correct. Temporal persistence is used as a candidate-selection heuristic. A true pavement defect can persist across nearby frames, but so can a systematic false detection such as a marking, vehicle feature, or other stable visual pattern. Conversely, a true defect can be missed in one or more frames because of motion blur, occlusion, illumination change, or detector instability. The method therefore aims to reduce transient/redundant candidates, not to replace label-quality assurance.

Figure 4.2 shows a real isolated error from the corrected unfiltered candidate set. A visually clear false Longitudinal Crack candidate appears in frame 100, while the immediately surrounding frames shown have no detector-positive candidate. The frame is present in the unfiltered candidate set but absent from the final filtered/subsampled arm. This example is

consistent with the intended temporal-selection behavior, although the final pipeline changes persistence filtering and representative-frame selection together.



Figure 4.2: Real consecutive-frame example of an isolated false detector response. The false Longitudinal Crack candidate appears only in the center frame of the displayed sequence and is absent from the final filtered/subsampled training arm. The example illustrates the behavior of the combined candidate-selection pipeline; it does not isolate the causal contribution of persistence filtering from representative-frame selection.

A complementary sequence is shown in Figure 4.3. Five consecutive Longitudinal Crack candidate frames occur while the vehicle passes through strong overpass shadow; the final arm retains frames 42, 44, and 46 as representative samples. This sequence demonstrates redundancy reduction under challenging illumination rather than claiming that shadow caused the detections.



Figure 4.3: Consecutive detector-positive frames during an overpass-shadow transition. The corrected unfiltered candidate set contains all five displayed frames, while the final filtered/subsampled arm retains frames 42, 44, and 46. The example demonstrates representative reduction within a temporally consistent sequence under challenging illumination.

4.3.2 Pipeline Stages

Four stages are applied in sequence to each input video.

Stage 1: Frame Extraction. Each input video is sampled at four frames per second, regardless of its native frame rate. This rate was selected empirically to provide enough temporal density for the tracker to reliably associate detections of the same feature across frames, without generating excessive near-duplicate images at low vehicle speeds. At typical inspection-vehicle speeds, four frames per second provides multiple captured frames for any surface feature that remains visible as the vehicle passes, while limiting near-duplicate images. The exact distance between consecutive sampled frames is speed/4 fps and varies with vehicle

speed; the four-fps rate was chosen empirically as a practical balance between temporal coverage and dataset size. Extracted frame filenames are prefixed with a batch tag derived from the source video folder name. This prefix is necessary because the consumer dashcam hardware used in this project reuses filenames across recording sessions—the same filename (e.g., `GRMN0015.MP4`) appears in multiple collection batches and refers to entirely different footage each time. Without the prefix, frames from different collection sessions stored in the same output directory would silently overwrite each other.

Stage 2: Object Tracking. Rather than running the detector independently on each frame, the pipeline runs YOLO11x in tracking mode using ByteTrack [15]. This assigns each detected object a persistent numerical identifier and attempts to keep that same identifier on the same physical object across consecutive frames. Running in tracking mode, rather than detection mode, is what makes it possible to ask the question central to temporal filtering: for any given detection, how many consecutive frames did the same physical object appear in?

Stage 3: Temporal-Persistence Filtering. For each tracked identifier, the pipeline finds all maximal runs of consecutive frame indices in which that identifier appears. A gap of at most one missed frame within a run is tolerated, to handle single-frame detection dropout caused by brief motion blur or partial occlusion. A qualifying run must contain at least three tracked observations, while allowing at most one missed sampled frame between successive observations. Runs that do not meet this criterion are discarded as more likely to represent transient detections than stable road features.

An implementation error in an early version of this stage is worth documenting because it illustrates a non-obvious pitfall in applying general-purpose trackers to this problem. ByteTrack reuses numerical identifiers once a tracked object has been lost for enough frames. An initial implementation counted the total number of frames in which a given identifier appeared across the entire video, which incorrectly merged multiple short, unrelated detections that happened to share a recycled identifier into a single, inflated occurrence count. The error was identified by plotting the spatial trajectory of the affected detections: a genuinely persistent, correctly tracked crack produces a bounding box that moves smoothly and predictably in the direction of vehicle travel, while the recycled-identifier case showed the bounding box position jumping abruptly and non-physically between frames. The implementation was corrected to require genuine frame-index contiguity—consecutive frame numbers, not aggregate occurrence count—before applying the minimum-length threshold.

Stage 4: Representative Subsampling. A qualifying run spanning many consecutive frames captures the same physical crack from nearly identical viewpoints, producing a collection of near-duplicate images. Adding all of these to the training set provides little additional information compared to a smaller, more diverse sample from the same run. Each qualifying run is therefore subsampled to three representative frames: the first, the middle, and the last. This preserves viewpoint variation within the run—approaching, directly overhead, and departing—while limiting the number of near-duplicate training images per physical crack.

Algorithm 1 summarizes the candidate-selection logic independently of implementation details.

Algorithm 1 Temporal candidate selection for one input video

Require: Ordered sampled frames $F_1, \ldots, F_T$ with tracked detections $(\text{id}, \text{class}, \text{box}, \text{confidence})$
Require: Minimum observed detections $L = 3$; maximum tolerated missed sampled frame $G = 1$
Ensure: Representative auto-labeled frames and labels

```
Group all detections by tracker identifier
for all track identifiers k do
    Sort detections for k by sampled-frame index
    Split the ordered detections into maximal runs such that successive observed frame indices differ by at most G + 1
    for all runs r for track k do
        if |r| < L then
            discard r
        else
            select the first, middle, and last observed frames of r
            write the corresponding image(s), YOLO label(s), and metadata
        end if
    end for
end for
return the union of selected representative frames across qualifying runs
```

The algorithm is applied to tracker observations rather than to the total lifetime count of a numeric track identifier. This distinction prevents unrelated detections from being merged when ByteTrack later reuses an identifier after the original object has been lost.

4.3.3 Implementation Notes

The complete framework is packaged as a single command-line tool (`auto_label_pipeline.py`). It accepts a video file, a directory of video files, or a directory of pre-extracted images

as input. When video files are provided, the full four-stage pipeline runs: frame extraction, ByteTrack tracking, temporal-persistence filtering, and subsampling. When a directory of pre-extracted images is provided, the tool runs the detector directly on each image and applies the same confidence threshold and output structure, without the tracking or persistence stages (since temporal context is not available for arbitrary image collections). Output is written to an automatically numbered directory (e.g., `01_GRMN0032.MP4_output`), containing:

- `images/`: retained frames, unmodified.
- `images_annotated/`: the same frames with bounding boxes, class labels, and confidence scores drawn for human review.
- `labels/`: YOLO-format label files (class ID, normalized x, y, w, h) ready for retraining.
- `manifest.json`: a per-detection record with source video, GPS coordinates, distress class, confidence score, and bounding-box diagonal in pixels.
- `metadata.csv`: a flat CSV version of the same information, one row per detection.

Table 4.2: Verified auto-labeling and final-training configuration used for reproducibility.

Component	**Recorded setting**
Video frame extraction	4 frames/s
YOLO tracking inference	image size 896; confidence floor 0.25; IoU 0.45; `bytetrack.yaml`
Temporal selection	minimum 3 tracked observations; maximum gap 1 sampled frame; first/middle/last representative frames
ByteTrack configuration	high threshold 0.25; low threshold 0.10; new-track threshold 0.25; track buffer 30; match threshold 0.80; score fusion enabled
Final training	100 epochs; image size 896; batch 8; seed 42; cosine LR; mosaic 1.0; mixup 0.1; HSV 0.015/0.7/0.4; perspective 0.001; scale 0.8
Final training software	Ultralytics 8.4.41; Python 3.10.20; PyTorch 2.11.0+cu130
Auto-label support software	OpenCV 4.13; NumPy 2.2.6; EasyOCR 1.7.2
Final GPU	NVIDIA RTX A4000 (15,976 MiB reported)

GPS coordinate extraction. The dashcam hardware used in this project burns coordinates, timestamp, and vehicle speed directly into the video frame as visible on-screen

text (Figure 4.4a), but does not produce a separate structured GPS file. Coordinates are recovered by cropping the text region from each retained frame and applying EasyOCR, a Python-native optical character recognition library that operates entirely within the existing Conda environment without requiring external system binaries.

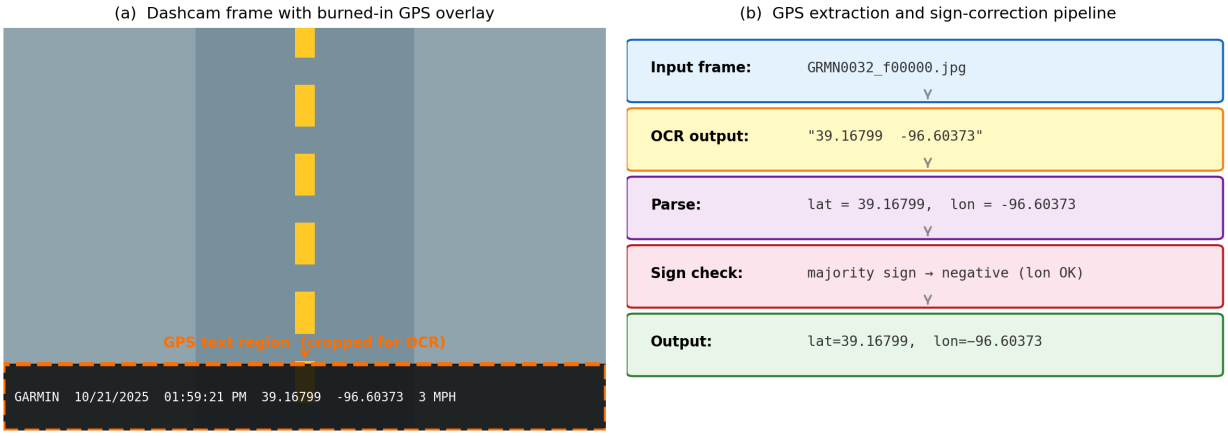


Figure 4.4: GPS coordinate extraction from the on-screen dashcam text overlay. (a) A representative frame showing the burned-in GPS text strip at the bottom of the image; the orange dashed box indicates the region cropped for OCR. (b) Processing steps from raw frame to validated GPS output, including the majority-sign correction applied to handle intermittent sign-inversion errors in the OCR output. The specific coordinate values shown (`39.16799, -96.60373`) are real values read from pipeline run `06_GRMN0032.MP4_output/metadata.csv`.

During development, an intermittent OCR failure was identified: the sign of the recognized longitude value was occasionally inverted on individual frames while all other digits were correct. The consequence was a physically implausible coordinate sequence in which the vehicle appeared to cross a longitude hemisphere boundary within a single one-minute video clip. Because a single source video represents one continuous drive, the sign cannot genuinely change. The fix was to determine the majority longitude sign across all successfully recognized frames in the video and correct any frame that disagreed. Every corrected coordinate is flagged in the output metadata (the `gps_sign_corrected` column in `metadata.csv`) rather than silently overwritten. The fix was validated against the real pipeline output in which 13 sign corrections were identified and applied in one video clip (`GRMN0032`), with post-correction coordinates confirmed to show a smooth, monotonically changing longitude sequence consistent with vehicle travel along a straight road.

Physical distress measurement. Each retained detection is associated with a bounding-box diagonal in pixels. Converting this pixel measurement to a real-world physical length (e.g., crack length in meters) requires camera calibration information—mounting height above the road, camera tilt angle, and horizontal field of view—that was not available for the

hardware used in this project. The pixel diagonal is retained in the output metadata as a placeholder; a field (`real_world_length_m`) is reserved in the manifest for a future calibrated value.

4.4 Iterative Retraining Workflow

The auto-labeled output is incorporated into retraining through a controlled dataset-versioning procedure. The public-data training, validation, and test partitions are held fixed across experimental conditions. Newly generated KDOT auto-labels are added only to the training partition; the validation and test sets are not augmented or reshuffled. This design prevents newly collected video frames from entering the evaluation partitions and permits the effect of the additional auto-labeled training data to be compared under a common evaluation protocol.

The retraining workflow consists of four steps.

1. **Merge.** Accepted auto-labeled KDOT images and YOLO-format labels are incorporated into a new training manifest. The merge procedure checks for image/label consistency and filename collisions before constructing the new dataset version.

2. **Preserve evaluation partitions.** The original public validation and test manifests remain unchanged. The validation set contains 9,866 entries and is used for model selection during training. The test set contains 9,867 entries and remains frozen until final evaluation.

3. **Retrain.** Each experimental condition is trained from the same original YOLO11x pretrained weights rather than from the checkpoint produced by another experimental condition. The model architecture, training hyperparameters, input resolution, random seed, augmentation configuration, and optimization procedure are held constant. Only the contents of the training manifest change between conditions.

4. **Evaluate.** The checkpoint selected using validation performance is evaluated with standalone `yolo detect val` on the identical frozen test manifest. No test-set result is used for checkpoint selection.

Dataset-integrity correction. A post-processing audit of the historical Cycle 1 artifacts identified that the originally reported 2,120 retained image/label pairs represented two filename variants of the same 1,060 underlying retained frames. The duplicate copies were byte-identical and therefore did not represent 2,120 independent training examples. The

final controlled experiment reported in this thesis uses the 1,060 unique filtered/subsampled frames.

The same audit also identified that an earlier unfiltered-label artifact contained six fields per annotation row because detector confidence had been retained in addition to the five fields required by the YOLO detection-label format. The final unfiltered condition was reconstructed using valid five-field labels. These corrected datasets were used for the final controlled comparison described below. Historical results produced before these corrections are treated as diagnostic development results and are not used as the primary evidence for the thesis conclusions.

4.5 Experimental Design

The final experiment compares four training conditions using a common public training base and identical frozen validation and test partitions.

1. **Control.** The fixed public training manifest contains 78,928 image/label pairs.

2. **Filtered/subsampled condition.** The same 78,928-entry public training manifest is augmented with 1,060 unique KDOT auto-labeled frames retained by the full video candidate-selection pipeline. This pipeline applies temporal-persistence filtering followed by representative-frame subsampling. The resulting training manifest contains 79,988 entries.

3. **Random1060 condition.** The same public base is augmented with 1,060 unique frames sampled uniformly without replacement from the corrected 3,266-frame KDOT detector-positive candidate pool (selection seed 2026). This equal-size random comparator contains 79,988 training images and controls for the amount of added KDOT data.

4. **Unfiltered condition.** The same 78,928-entry public training manifest is augmented with 3,266 KDOT detector-positive frames from the same underlying footage and detector output, without the temporal-persistence and representative-subsampling stages. The resulting training manifest contains 82,194 entries.

The filtered/subsampled condition therefore uses 2,206 fewer KDOT training frames than the unfiltered condition, corresponding to a 67.5% reduction in the amount of added KDOT training imagery.

The equal-size random control separates the effect of the selected candidate set from the effect of adding the same number of KDOT frames, but it does not isolate temporal persistence from representative subsampling. The comparison should not be interpreted as

an ablation of temporal persistence alone. Two candidate-selection components change together: temporal-persistence filtering and representative-frame subsampling. The experiment therefore evaluates the complete filtered/subsampled candidate-selection strategy against retaining all detector-positive frames. It can determine whether the substantially smaller selected training set maintains comparable downstream detection performance, but it cannot attribute any observed difference uniquely to temporal persistence.

All four conditions use the same YOLO11x architecture and the same original pretrained initialization. Training was performed for up to 100 epochs with an input resolution of 896 pixels, batch size 8, random seed 42, cosine learning-rate scheduling, and the same data-augmentation configuration. Periodic checkpoints were retained during the final runs for recovery and integrity verification.

All four training runs reached their 100-epoch target; the Random1060 run was resumed following a scheduler interruption. Checkpoint selection was performed using the common 9,866-entry validation set. In the installed Ultralytics implementation, model fitness for detection is determined by validation mAP at IoU thresholds 0.50–0.95 (mAP50–95). The selected checkpoints occurred at epoch 93 for the control condition, epoch 91 for the filtered/subsampled condition, epoch 92 for the unfiltered condition, and a validation-selected checkpoint for the Random1060 run. The Random1060 checkpoint epoch is not independently established in the audit summary. The frozen 9,867-entry test set was not used for model selection.

The final comparison consequently differs from earlier development-stage evaluations in two important respects: the KDOT additions are corrected and deduplicated, and the four training conditions share identical public training, validation, and test manifests. The final experiment is therefore used as the primary quantitative comparison throughout the remainder of this thesis.

Lane-marking and candidate-review analysis. Painted lane markings are a potential failure mode because they can resemble longitudinal pavement cracks and remain visible across consecutive frames. A historical Cycle 1 checklist contains 50 image-level candidate-review records, including 30 associated with the Longitudinal Crack category and 20 from other categories. The recorded decisions are `Keep`, `Unsure`, and `other_fp`. These screening decisions are not treated as independently verified detection-level true-positive or false-positive annotations. Their interpretation and the status of the separate temporal-filtering audit are reported in Sections 5.6 and 5.7.

Throughput analysis. Wall-clock execution time was recorded from SLURM job logs and compared with the duration of source footage. The resulting ratio answers the throughput question as an observed end-to-end pipeline rate under the available shared GPU-cluster environment; it is not a controlled microbenchmark of individual processing stages or a universal real-time guarantee.

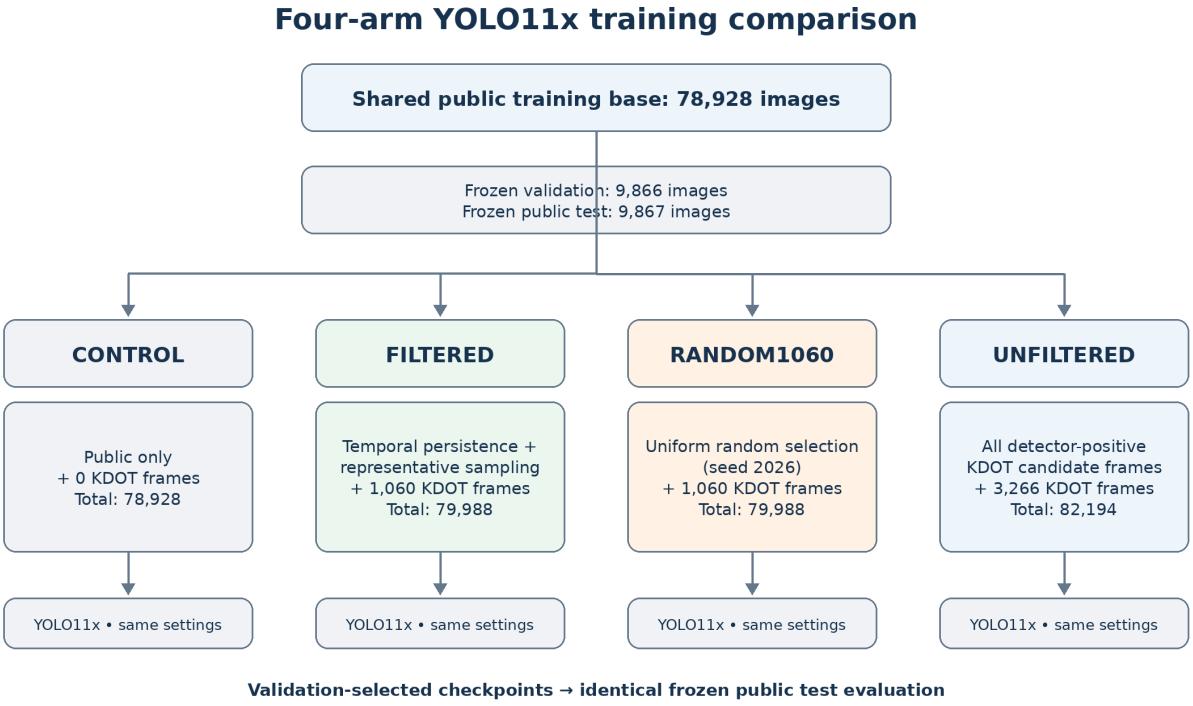


Figure 4.5: Four-arm experimental design for the final YOLO11x comparison. All conditions share the same 78,928-image public training base and frozen validation (9,866 images) and test (9,867 images) manifests. The filtered and Random1060 conditions each add 1,060 KDOT frames, whereas the unfiltered condition adds 3,266. KDOT imagery is used for training only. All four conditions use the same model architecture and training configuration, with checkpoints selected using validation performance before evaluation on the identical frozen public test set.

Chapter 5 – Results

The final results reported in this chapter supersede the historical Cycle 1 comparison produced before the dataset-integrity audit. All four final models were evaluated on the identical frozen public test manifest containing 9,867 images. The selected `best.pt` checkpoint for each model was chosen using validation performance only. The final test jobs completed successfully with exit code `0:0`: job 11672661 for the control model, job 11672662 for the filtered/subsampled model, job 11672663 for the unfiltered model, and job 12448051 for the Random1060 model.

5.1 Final Frozen-Test Aggregate Performance

Table 5.1: Final four-arm performance on the identical frozen 9,867-image public test set.

Training condition	Precision	Recall	mAP50	mAP50–95
Control (public only)	0.681	0.611	0.665	0.383
Filtered/subsampled (+1,060 KDOT)	0.675	0.615	0.664	0.381
Random1060 (+1,060 KDOT)	0.673	0.610	0.661	0.381
Unfiltered (+3,266 KDOT)	0.680	0.614	0.667	0.383

Relative to the control, the filtered/subsampled condition changed precision by −0.006, recall by +0.004, mAP50 by −0.001, and mAP50–95 by −0.002. The unfiltered condition changed precision by −0.001, recall by +0.003, mAP50 by +0.002, and mAP50–95 by approximately 0.000 at the reported precision. The filtered/subsampled and unfiltered conditions differ by 0.003 in mAP50 and 0.002 in mAP50–95.

These differences are small. The experiment used one training seed per condition and did not include repeated-seed uncertainty estimation or statistical significance testing. The final numbers should therefore be interpreted as observed one-run comparisons rather than evidence that any condition is statistically superior.

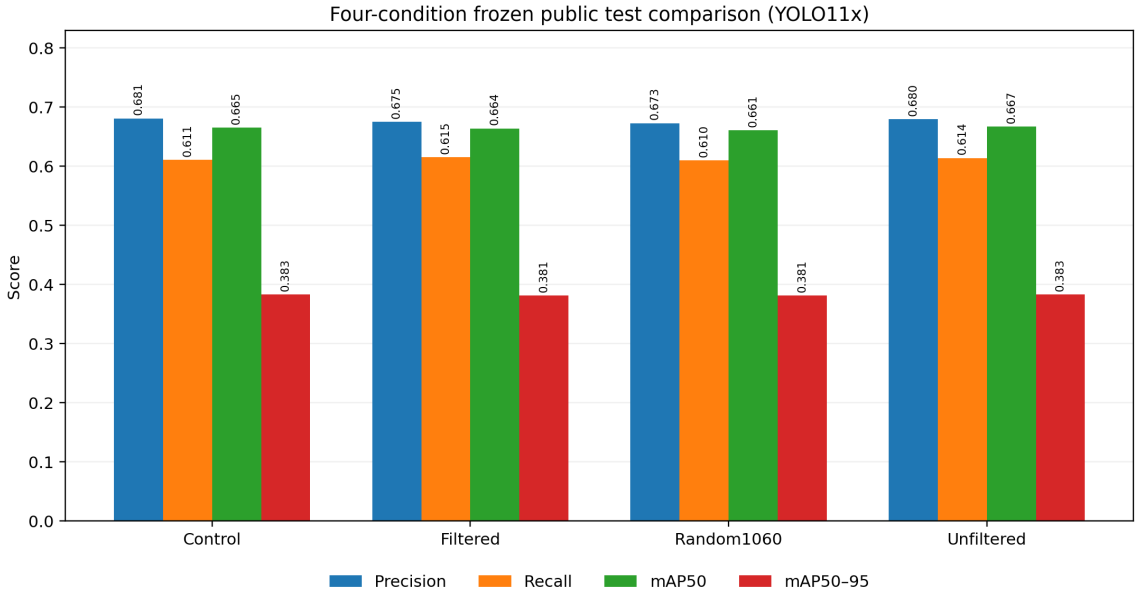


Figure 5.1: Final frozen-public-test comparison of four YOLO11x training conditions: public-only control, filtered/subsampled KDOT augmentation, equal-size random KDOT sampling (Random1060), and corrected unfiltered KDOT augmentation. All models were evaluated on the same 9,867-image public test set. The observed differences do not establish statistically significant superiority.

5.2 Per-Class Frozen-Test Performance

Table 5.2: Per-class mAP50 on the frozen public test set for all four conditions. Δ_F denotes filtered/subsampled minus control.

Class	Control	Filtered	Random	Unfiltered	Δ_F
Longitudinal Crack	0.620	0.619	0.617	0.619	−0.001
Transverse Crack	0.719	0.716	0.721	0.727	−0.003
Alligator Crack	0.681	0.683	0.678	0.678	+0.002
Pothole	0.627	0.623	0.623	0.630	−0.004
Patch	0.879	0.872	0.879	0.877	−0.007
Block Crack	0.467	0.469	0.449	0.470	+0.002

Across the six classes, the filtered/subsampled condition differs from the control by between −0.007 and +0.002 mAP50, while the unfiltered condition differs from the control by between −0.003 and +0.008. No class shows a large performance collapse under either augmented training condition.

The corresponding mAP50–95 values for control/filtered/unfiltered were 0.337/0.334/0.333 for Longitudinal Crack, 0.349/0.347/0.355 for Transverse Crack, 0.389/0.391/0.388 for Alligator Crack, 0.334/0.334/0.332 for Pothole, 0.658/0.651/0.656 for Patch, and 0.229/0.232/0.232

for Block Crack. These per-class results reinforce the aggregate finding that all four YOLO11x training conditions achieved broadly comparable performance under the stricter IoU-averaged metric. The Random1060 per-class mAP50–95 values are reported in Section 5.4.

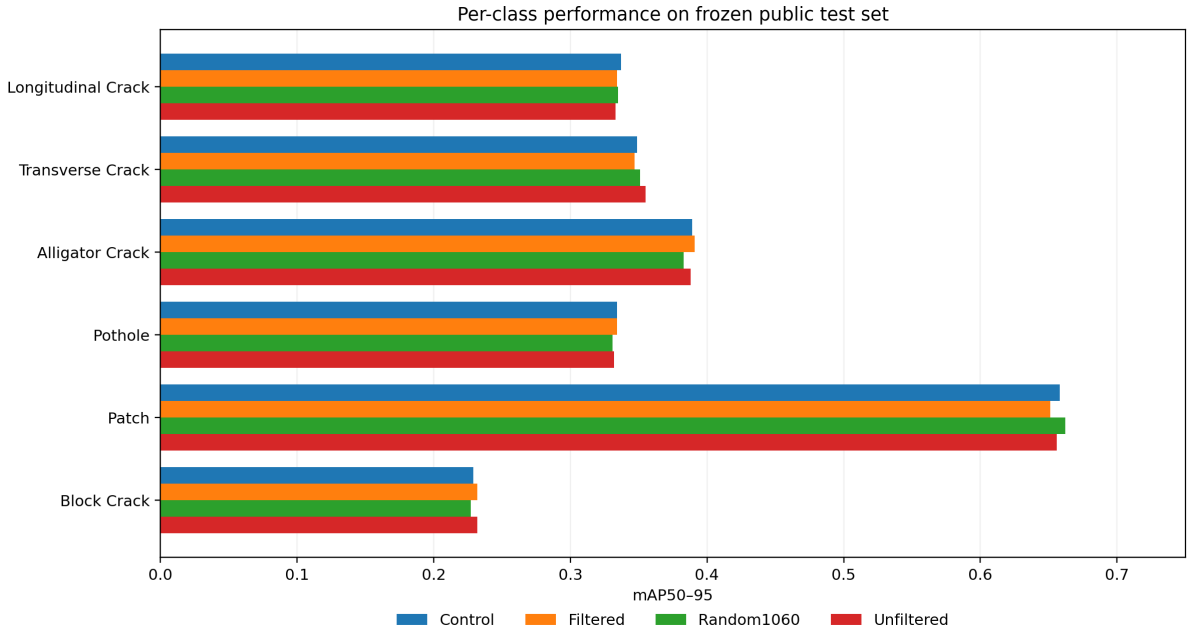


Figure 5.2: Per-class mAP50–95 on the identical frozen public test set. No class shows a large collapse after reducing the KDOT augmentation from 3,266 unfiltered detector-positive frames to 1,060 filtered/subsampled frames.

5.3 Filtered/Subsampled vs. Unfiltered Candidate Volume

The filtered/subsampled condition adds 1,060 unique KDOT frames, whereas the unfiltered condition adds 3,266 detector-positive KDOT frames from the same underlying candidate pool. The combined filtering and representative-subsampling strategy therefore removes 2,206 candidate frames, a 67.5% reduction in added KDOT training-frame volume.

Table 5.3: Training-data volume and final frozen-test performance for the four controlled conditions.

Condition	Training images	mAP50	mAP50–95
Control	78,928	0.665	0.383
Filtered/subsampled	79,988	0.664	0.381
Random1060	79,988	0.661	0.381
Unfiltered	82,194	0.667	0.383

The filtered/subsampled model therefore reaches essentially the same frozen-test performance range while using substantially fewer added KDOT frames. This is a data-efficiency result, not evidence that filtering improves detector accuracy. The experiment also does not isolate temporal persistence alone because temporal-persistence filtering and representative-frame subsampling were changed together.

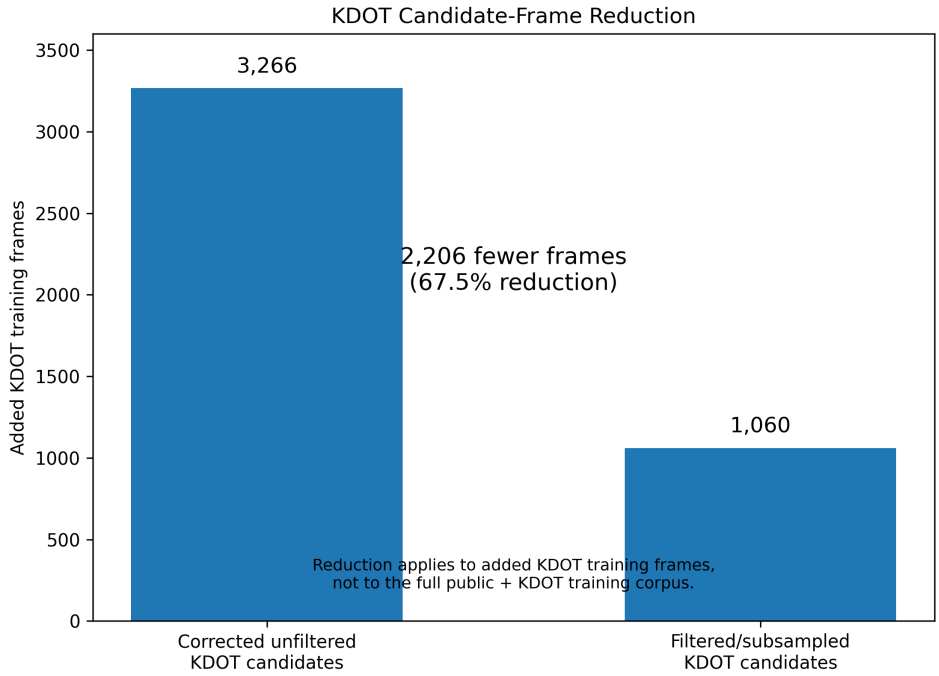


Figure 5.3: Reduction in added KDOT training-frame volume. The combined temporal-persistence and representative-frame strategy reduces the detector-positive candidate set from 3,266 to 1,060 frames, or 67.5%. This percentage applies to the added KDOT frames, not to the entire public-plus-KDOT training corpus.

5.4 Equal-Size Random Selection Control

The Random1060 comparator sampled 1,060 distinct KDOT frames uniformly without replacement from the corrected 3,266-frame detector-positive pool, with selection seed 2026. It used the same 78,928-image public training base, yielding the same total of 79,988 training images as the filtered/subsampled condition. Random1060 achieved precision 0.673, recall 0.610, mAP50 0.661, and mAP50–95 0.381 on the common frozen test set. The filtered condition was 0.003 higher in mAP50 and equal to three reported decimals in mAP50–95. The single-seed result does not demonstrate statistically reliable superiority of temporal selection over random frame selection; rather, the equal-size comparison provides a more appropriate reference for interpreting data-selection choices.

The per-class Random1060 mAP50 results were 0.617 (Longitudinal Crack), 0.721 (Trans-

verse Crack), 0.678 (Alligator Crack), 0.623 (Pothole), 0.879 (Patch), and 0.449 (Block Crack). Its per-class mAP50–95 values were 0.335, 0.351, 0.383, 0.331, 0.662, and 0.227, respectively, in the same class order. The Random1060 evaluation was completed under SLURM job 12448051 using the validation-selected `best.pt` checkpoint.

5.5 Final-Test Dataset Behavior

Each final evaluation processed the same 9,867 test images and 18,137 detection boxes. Ultralytics emitted the same warning in all four runs: the dataset contained 85 segment records while the detection loader found 18,137 boxes, so segment geometry was removed and boxes were used for detection evaluation. Because the warning and effective test annotations were identical across all four conditions, this mixed-format issue does not create a differential advantage among the four models. It remains a dataset-quality limitation and is reported explicitly.

The per-class ground-truth box counts used by the final evaluator were 8,004 Longitudinal Crack instances, 4,163 Transverse Crack instances, 2,112 Alligator Crack instances, 1,108 Pothole instances, 916 Patch instances, and 1,834 Block Crack instances.

5.6 Final Dataset Integrity and Manual-Audit Status

An integrity audit of the public corpus confirmed 78,928 training, 9,866 validation, and 9,867 test image entries, with no duplicate manifest paths, no broken image symbolic links, and no resolved source-file paths shared across splits. Candidate images with matching file sizes across different splits (17,415 files; approximately 0.856 GiB) were SHA-256 hashed. Exactly one byte-identical train–validation pair was found: `6.Uncracked_06052.jpg` in training and `6.Uncracked_08518.jpg` in validation. Both label files are empty. No exact byte-identical images were identified in train–test or validation–test comparisons. This is an identified train–validation leakage instance; exact-content hashing does not rule out resized, recompressed, or near-duplicate imagery. No files were removed from the completed benchmark.

The source validation partition contained 91 polygon-style annotation rows across 57 files, and the test partition contained 85 such rows across 48 files, in addition to standard five-field detection labels. Ultralytics used bounding boxes in detection evaluation. The validation loader also excluded one image with an invalid negative annotation coordinate, giving 9,865 effective validation images from the 9,866-image manifest. A corresponding invalid-coordinate warning was recorded for one training image. These are shared-data quality limitations rather than alterations made after evaluation.

A separate temporal-filtering review sample consists of 40 KDOT frames and 75 predicted boxes (20 retained-group frames and 20 excluded-group frames). At the time of this audit, all

75 saved `manual_verdict` fields and all 40 frame-level review statuses remained incomplete. Earlier aggregate TP/FP/uncertain notes could not be reconciled to individual box records and are therefore not reported as verified results. The earlier Cycle 1 image-level checklist is distinct and must not be interpreted as box-level correctness evidence for the final temporal-filtering experiment.

5.7 Targeted Lane-Marking Review

The historical Cycle 1 review checklist contains 50 image-level records, with 46 marked `Keep`, two marked `Unsure`, and two marked `other_fp`. These decisions document candidate-image screening rather than independently verified detection-level true positives and false positives.

Consequently, the `Keep` count must not be interpreted as a measured detector precision, and the checklist does not support a population lane-marking false-positive rate. The separate 40-frame, 75-box temporal-filtering audit described in Section 5.6 remains incomplete. Qualitative examples of visually identified failure modes are presented in the following section.

5.8 Qualitative Auto-Label Failure Analysis

The historical 50-image Cycle 1 screening review described in Section 5.7 provides limited qualitative context but does not constitute a comprehensive detection-level error analysis. A later manual review of corrected unfiltered candidates identified visually clear errors involving road paint/lane markings and a vehicle-associated false Patch prediction. Figure 5.4 shows three representative cases using the actual corrected YOLO boxes.



Figure 5.4: Representative manually reviewed auto-label failures. A false Patch prediction is placed on an adjacent vehicle and remains in the final filtered/subsampled arm; two road-paint/lane-marking crack confusions are present in the corrected unfiltered candidates but absent from the final arm. These examples show that candidate selection can remove some erroneous frames but does not guarantee correctness of retained boxes.

One especially important limitation is frame `GRMN0039_f00044`: the corrected unfiltered

label and the final filtered label contain the same class-4 box, visually judged during review as a false Patch prediction on the adjacent vehicle. Figure 5.5 presents this case directly. It demonstrates why the temporal stage should be understood as candidate-volume reduction rather than automatic label validation.



Figure 5.5: False prediction that survives candidate selection. The same Patch box is present in the corrected unfiltered candidate set and the final filtered/subsampled arm. Temporal persistence and representative selection therefore do not provide a box-level correctness guarantee, and retained candidates still require quality assurance.

The contrast between Figures 4.2 and 5.5 is central to interpreting the method. An isolated false response can be excluded by the final selection procedure, while a stable or repeatedly supported false response can remain. The controlled quantitative experiment evaluates the downstream effect of selecting fewer frames; it does not establish that every retained pseudo-label is correct.

5.9 Pipeline Throughput

One historical filtered run processed 13,344 sampled frames in 32.5 minutes, corresponding to approximately 55.9 minutes of video-input workload at four sampled frames per second and an observed footage-to-compute ratio of approximately 1.72×. A second unfiltered run processed 6,672 sampled frames in 11.5 minutes, corresponding to approximately 27.9 minutes of input and an observed ratio of approximately 2.43×.

These throughput values are practical observations from shared-cluster jobs, not controlled benchmarks. In addition, the later dataset-integrity audit showed that the historical filtered artifact contained duplicated retained imagery. The throughput figures therefore describe processing workload, not unique roadway coverage or unique training examples. The 1,060 unique filtered frames used in the final experiment should be used when discussing final training-data volume.

Chapter 6 – Discussion

6.1 Core Finding: Data Reduction with Comparable Public-Test Performance

The equal-size random result provides an additional reference: filtered mAP50 was 0.664 versus random mAP50 0.661, while both reported mAP50–95 0.381. The final controlled experiment supports a narrower and more defensible conclusion than the historical Cycle 1 results suggested. The filtered/subsampled training condition did not improve aggregate frozen-test accuracy relative to the control: mAP50 changed from 0.665 to 0.664 and mAP50–95 changed from 0.383 to 0.381. The unfiltered condition produced mAP50 0.667 and mAP50–95 0.383. All four models therefore fall within a narrow performance band on the common public test set.

The more substantial experimental difference is training-data volume. The filtered/subsampled strategy added 1,060 unique KDOT frames, whereas the unfiltered strategy added 3,266. The combined selection procedure therefore reduced the KDOT augmentation volume by 67.5% while the resulting model remained within 0.003 mAP50 and 0.002 mAP50–95 of the unfiltered model. Relative to the control, the filtered model differed by only −0.001 mAP50.

The near-zero metric change is itself informative for the data-selection problem. The unfiltered condition supplied 2,206 additional detector-positive KDOT frames, yet those extra frames changed the frozen-public-test result by only 0.003 mAP50 and 0.002 mAP50–95 relative to the filtered/subsampled condition. Under this evaluation, the marginal signal provided by retaining those additional, highly correlated video frames was therefore small relative to the added storage, review, and retraining volume. This interpretation is deliberately limited to the frozen public-domain test distribution; it does not establish that the removed frames would have equally low value on an independently annotated KDOT-domain test set.

Operationally, if a smaller selected candidate set produces approximately the same downstream performance as retaining every detector-positive frame, storage, later quality-control effort, and retraining input volume can be reduced without a large observed penalty on the evaluation used here.

6.2 What the Comparison Does and Does Not Isolate

The filtered/subsampled condition changes two stages simultaneously: temporal-persistence filtering and representative first/middle/last frame subsampling. Consequently, the 67.5%

reduction cannot be attributed uniquely to temporal persistence. A stricter component-level ablation would require additional conditions, such as persistence filtering with every frame from qualifying tracks retained, and representative subsampling applied without a persistence requirement. Those conditions were not part of the final experiment.

6.3 Interpretation of the Small Metric Differences

The aggregate changes are too small to support a strong accuracy ranking. The filtered condition is 0.001 below the control in mAP50, while the unfiltered condition is 0.002 above it. The filtered and unfiltered models differ by 0.003. Similar small differences appear in mAP50–95 and at the class level.

Because each condition was trained once with a single random seed, the experiment does not estimate run-to-run variance. No confidence interval or statistical significance test can therefore be attached to the differences among the four trained models. The appropriate interpretation is descriptive: the models produced very similar performance under the common evaluation, and the filtered condition achieved that range with substantially fewer added KDOT frames.

6.4 Domain Generalization and the Meaning of the Frozen Test Set

The frozen test set is composed of the public remapped corpus rather than an independently held-out KDOT roadway-video test set. This design is valuable for comparing the four training conditions under exactly the same evaluation distribution, but it places an important limit on the conclusions.

The experiment shows that adding either 1,060 filtered/subsampled or 3,266 unfiltered KDOT pseudo-labeled frames did not produce a large change in performance on the common public-domain test set. It does not directly measure whether either augmented model is better on future KDOT footage. A separate KDOT-domain held-out test set with independent human ground truth would be required to make that claim.

6.5 Failure Modes and the Role of Quality Assurance

Painted lane markings and other stable non-distress structures are potential failure modes because they can resemble longitudinal pavement cracks and persist across consecutive frames. The historical Cycle 1 checklist documented 50 image-level screening decisions, but it did not provide independently verified bounding-box correctness labels. Consequently, those records cannot establish a lane-marking false-positive rate. The separate qualitative review identified

visually apparent road-paint and lane-marking confusions among the corrected unfiltered candidates. These observations illustrate possible detector errors without supporting a quantitative estimate of their frequency.

The retained false Patch-on-vehicle example further shows that temporal persistence is not equivalent to semantic correctness. A false prediction can remain stable enough to survive the candidate-selection workflow. Accordingly, the method should be deployed as a mechanism for reducing redundancy and transient detector responses before review, not as a substitute for review itself. This distinction is especially important if later retraining cycles are expected to reuse pseudo-labeled data repeatedly, because systematic errors could otherwise be reinforced.

6.6 Practical Implications for KDOT

For KDOT, the strongest result is that the auto-labeling pipeline can turn routine dashcam footage into a substantially smaller retraining candidate set while preserving comparable public-test performance. In the final experiment, the filtered/subsampled strategy reduced the added KDOT frame count from 3,266 to 1,060. This reduction can lower storage and review burden and limit the number of near-duplicate frames passed to retraining.

The pipeline also preserves inspectable outputs: retained images, annotated review copies, YOLO labels, and structured metadata can be examined before they are incorporated into a future training cycle. GPS coordinates recovered from the dashcam overlay provide a route toward location-aware maintenance reporting, although the OCR output was not independently validated against a ground-truth GPS trace.

6.7 Threats to Validity

Internal validity. The final control, filtered/subsampled, and unfiltered runs use common public training data, identical frozen validation and test manifests, the same original YOLO11x pretrained initialization, and the same recorded training configuration. The test set was not used for final checkpoint selection. However, the four conditions were not repeated across multiple random seeds, so small metric differences cannot be separated from ordinary training stochasticity. The filtered/subsampled condition also changes two candidate-selection components together.

Dataset validity. The public corpus merges datasets with different acquisition viewpoints, image resolutions, class definitions, and annotation conventions. A content-hash audit also identified one duplicated background image across the public training and validation partitions, although no exact train–test or validation–test duplicates were found. The audit did not test

perceptually similar transformed images. One invalid negative-coordinate annotation caused a validation image to be excluded; a training image was also flagged for an invalid coordinate. During final evaluation, Ultralytics detected 85 segment records alongside 18,137 boxes and removed segment geometry because the dataset was being evaluated as a detection dataset. This behavior was common to all four conditions, but it indicates that the source corpus is not perfectly format-homogeneous.

Construct and measurement validity. Bounding boxes represent approximate distress location and extent rather than pixel-accurate crack geometry or physical crack dimensions. The earlier lane-marking review used a targeted sample. The distinct 40-frame/75-box temporal-filtering manual audit remains incomplete and its provisional aggregate counts are not treated as verified accuracy evidence. GPS values were recovered through OCR and were not independently compared with a ground-truth positioning trace.

External validity. The final quantitative test is public-domain rather than an independently annotated KDOT-domain test. The KDOT augmentation footage came from a limited collection setting and one primary dashcam configuration. The results therefore do not establish generalization to all future KDOT roadways, seasons, lighting conditions, pavement materials, camera systems, or vehicle speeds.

Model-selection history. YOLO11x was selected during an earlier development-stage comparison of YOLO11 model scales. The archived development record does not establish with sufficient certainty that this architecture-scale choice was made without reference to held-out performance. This does not alter the controlled comparison among the four final YOLO11x conditions, but it limits interpretation of the frozen test set as completely untouched by every development decision made during the broader project.

6.8 Limitations

The thesis evaluates one final augmentation experiment rather than repeated self-training cycles. It does not include an independently annotated KDOT ground-truth test set or repeated-seed training. The filtered/subsampled comparison cannot distinguish the individual effects of temporal persistence and representative subsampling because both stages change together.

Candidate selection is also not a box-level correctness guarantee. The qualitative failure analysis includes an erroneous Patch prediction that survives into the final filtered/subsampled arm. Conversely, some removed frames may contain valid distress that was not retained

because the detector/tracker did not establish a qualifying run or because representative subsampling selected different views. The present experiment measures downstream training behavior, not pseudo-label precision/recall against an independently annotated KDOT video ground truth.

The final quantitative evaluation is performed on the frozen public-domain test set. KDOT imagery enters training only, so the results do not establish target-domain generalization to future KDOT footage. The system also does not convert bounding boxes into physical crack measurements because camera-calibration parameters were unavailable. GPS extraction remains OCR-based and was not independently validated against a survey-grade positioning trace. These limitations position the system as a tested research prototype for dataset growth and roadway-video processing rather than a complete autonomous pavement-condition measurement system.

6.9 Data and Code Availability

The thesis uses a mixture of public pavement-distress datasets and project-generated KDOT processing outputs. The curated multi-source project dataset is published at

`https://huggingface.co/datasets/Deeksha9/pavement-distress-detection`

Trained model weights are archived at

`https://huggingface.co/Deeksha9/pavement-distress-detection`

Table 3.1 separately documents which local source identities could and could not be traced defensibly to public upstream releases.

The final controlled experiment is associated with frozen dataset manifests, archived model checkpoints, SLURM evaluation jobs, launcher hashes, and checkpoint hashes. Distribution of raw KDOT roadway footage or other sponsor-provided artifacts should follow the applicable KDOT and Kansas State University release requirements. Appendix A records final evaluation provenance, while Appendix B incorporates the project software/user guide.

Chapter 7 – Conclusion

7.1 Summary

This thesis developed a reproducible pavement-distress auto-labeling and retraining system for agency-style roadway imagery. The operational pipeline combines YOLO11x detection, ByteTrack-based temporal association, temporal-persistence filtering, representative-frame subsampling, GPS-oriented metadata extraction, human-review artifacts, and controlled retraining. The final experiment serves as a validation of the pipeline's candidate-selection strategy rather than as a claim that temporal filtering increases detector accuracy.

The final controlled experiment compared four models on the same frozen 9,867-image public test set. The public-only control achieved precision 0.681, recall 0.611, mAP50 0.665, and mAP50–95 0.383. Adding 1,060 unique filtered/subsampled KDOT frames produced precision 0.675, recall 0.615, mAP50 0.664, and mAP50–95 0.381. An equal-size random KDOT sample achieved precision 0.673, recall 0.610, mAP50 0.661, and mAP50–95 0.381. Adding all 3,266 detector-positive KDOT frames produced precision 0.680, recall 0.614, mAP50 0.667, and mAP50–95 0.383.

The combined filtered/subsampled strategy therefore reduced the amount of added KDOT training imagery by 67.5% relative to the unfiltered condition while remaining within 0.003 mAP50 and 0.002 mAP50–95 of it. Relative to the control, the filtered model differed by −0.001 mAP50. With a single training seed and no significance testing, these small differences should not be interpreted as an accuracy advantage for any condition. The supported conclusion is instead that the much smaller filtered/subsampled candidate set preserved approximately comparable performance under the common public-domain evaluation.

7.2 Contributions

1. A command-line auto-labeling pipeline combining YOLO11x, ByteTrack, temporal-persistence filtering, representative-frame subsampling, GPS-overlay OCR, review artifacts, and YOLO-format retraining output.

2. An audited six-class pavement-distress corpus containing 98,661 retained images and 182,308 harmonized bounding boxes, with explicit local source/resolution accounting and a verified 15-to-6 label remapping.

3. A corrected controlled four-arm experiment using fixed public training, validation, and test populations, with KDOT imagery introduced into training only.

4. A data-efficiency result reducing the added KDOT candidate set from 3,266 to 1,060 frames (67.5%) while maintaining approximately comparable frozen-public-test detector performance.

5. A real failure analysis demonstrating both successful exclusion of an isolated false response and survival of a persistent false prediction, establishing that temporal selection reduces candidate volume but does not guarantee annotation correctness.

6. A reproducibility and operational-handoff record including final manifest/checkpoint hashes, software and training settings, evaluation job IDs, and one authoritative user and operations manual shared with the project repository and KDOT handoff.

7.3 Future Work

The highest-priority next experiment is construction of an independently annotated KDOT-domain test set that is never used for pseudo-label generation or training. Such a set would make it possible to determine whether KDOT pseudo-labeled augmentation improves performance on the target operational domain rather than only preserving performance on the public test distribution.

A second priority is repeated-seed training for the control, filtered/subsampled, Random1060, and unfiltered conditions. Replicated runs would quantify ordinary training variability and allow uncertainty intervals to be reported for the small differences observed here.

The equal-size Random1060 comparison has now been completed, but a component-level ablation should also separate temporal-persistence filtering from representative-frame subsampling. Additional conditions could retain every frame from qualifying persistent tracks or apply representative subsampling without a persistence requirement.

For deployment-oriented development, camera calibration would enable conversion from image-space bounding-box measurements to physical distress dimensions. GPS extraction should be validated against a ground-truth positioning source, and pseudo-label quality-control sampling should be expanded beyond the targeted lane-marking review. Finally, multiple sequential retraining cycles should be evaluated to determine whether pseudo-label errors accumulate, stabilize, or diminish as the dataset grows.

7.4 Closing Remarks

The principal contribution of this thesis is the implemented and documented auto-labeling system: a complete path from roadway video to temporally selected, reviewable, GPS-associated training candidates and reproducible retraining. The final experiment does not

show that temporal filtering increases detector accuracy. Instead, it validates a narrower operational benefit: the combined temporal-persistence and representative-subsampling strategy reduced the added KDOT candidate-frame volume by more than two thirds while the resulting model remained in essentially the same performance range as both the public-only control and the much larger unfiltered augmentation on the common public test set.

Within the limitations of a single seed, one augmentation experiment, and the absence of an independent KDOT-domain test set, the evidence supports candidate selection as a practical data-management mechanism for scalable pavement-monitoring workflows. It also establishes a reproducible software and evaluation baseline for future work on target-domain generalization, repeated retraining cycles, component-level ablation, and physical distress measurement.

References

[1] Transportation Research Board, Transportation Research International Documentation (TRID), "Use of Advanced Data Capture Tools on Measurements of Crack Lengths and Potholes for Estimates and Final Quantities," K-TRAN Project KSU-26-4, Kansas Department of Transportation and Kansas State University, active research project record. https://trid.trb.org/View/2652613

[2] Kansas Department of Transportation and Kansas State University, *Final Research Proposal for K-TRAN KSU-26-4 / KDOT Project RE-0922-01: Use of Advanced Data Capture Tools on Measurements of Crack Lengths and Potholes for Estimates and Final Quantities*, signed final research proposal, March 2025.

[3] H. Maeda, Y. Sekimoto, T. Seto, T. Kashiyama, and H. Omata, "Road Damage Detection and Classification Using Deep Neural Networks with Smartphone Images," *Computer-Aided Civil and Infrastructure Engineering*, vol. 33, no. 12, pp. 1127–1141, 2018. https://doi.org/10.1111/mice.12387

[4] Q. Zou, Z. Zhang, Q. Li, X. Qi, Q. Wang, and S. Wang, "DeepCrack: A Deep Hierarchical Feature Learning Architecture for Crack Segmentation," *Neurocomputing*, vol. 338, pp. 139–153, 2019. https://doi.org/10.1016/j.neucom.2019.01.036

[5] D. Arya, H. Maeda, S. K. Ghosh, D. Toshniwal, et al., "RDD2022: A Multi-National Image Dataset for Automatic Road Damage Detection," *Geoscience Data Journal*, 2024. https://doi.org/10.1002/gdj3.260 Data repository: https://doi.org/10.6084/m9.figshare.21431547.v1

[6] M. Ren, X. Zhang, X. Zhi, Y. Wei, Z. Feng, et al., "An Annotated Street View Image Dataset for Automated Road Damage Detection," *Scientific Data*, vol. 11, article 407, 2024. https://doi.org/10.1038/s41597-024-03263-7

[7] J. He, L. Gong, C. Xu, P. Wang, Y. Zhang, O. Zheng, G. Su, Y. Yang, J. Hu, and Y. Sun, "HighRPD: A High-Altitude Drone Dataset of Road Pavement Distress," *Data in Brief*, vol. 59, article 111377, 2025. https://doi.org/10.1016/j.dib.2025.111377 Dataset: https://data.mendeley.com/datasets/syswswj7djj/1

[8] M. Rezaeimanesh, A. Golroo, M. S. Fahmani, M. J. Amani, F. Hasanitabaar, M. S. Entezari, and S. Karimi, "Attain: Inclusive Annotated Pavement Distress Types and Severity Dataset," *Data in Brief*, vol. 61, article 111715, 2025. https://doi.org/10.1016/j.dib.2025.111715 Dataset: https://data.mendeley.com/datasets/nykrzdm74f/1

[9] J. Park, J. H. Lee, and J. Bang, "PotholeEye+: Deep-Learning Based Pavement Distress Detection System toward Smart Maintenance," *Computer Modeling in Engineering &*

Sciences, vol. 127, no. 3, pp. 965–976, 2021. https://doi.org/10.32604/cmes.2021.014669

[10] M. H. Asad, S. Khaliq, M. H. Yousaf, M. O. Ullah, and A. Ahmad, "Pothole Detection Using Deep Learning: A Real-Time and AI-on-the-Edge Perspective," *Advances in Civil Engineering*, article 9221211, 2022. https://doi.org/10.1155/2022/9221211

[11] S. Huang, W. Tang, G. Huang, L. Huangfu, and D. Yang, "Weakly Supervised Patch Label Inference Networks for Efficient Pavement Distress Detection and Recognition in the Wild," *IEEE Transactions on Intelligent Transportation Systems*, 2023. https://doi.org/10.1109/TITS.2023.3245192

[12] Q. Zhou, C. Yu, Z. Wang, Q. Qian, and H. Li, "Instant-Teaching: An End-to-End Semi-Supervised Object Detection Framework," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4081–4090, 2021.

[13] Y.-C. Liu, C.-Y. Ma, Z. He, C.-W. Kuo, K. Chen, P. Zhang, B. Wu, Z. Kira, and P. Vajda, "Unbiased Teacher for Semi-Supervised Object Detection," arXiv:2102.09480, 2021. https://arxiv.org/abs/2102.09480

[14] M. Xu, Z. Zhang, H. Hu, J. Wang, L. Wang, F. Wei, X. Bai, and Z. Liu, "End-to-End Semi-Supervised Object Detection With Soft Teacher," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 3060–3069, 2021.

[15] Y. Zhang, P. Sun, Y. Jiang, D. Yu, F. Weng, Z. Yuan, P. Luo, W. Liu, and X. Wang, "ByteTrack: Multi-Object Tracking by Associating Every Detection Box," in *European Conference on Computer Vision*, pp. 1–21, 2022. https://doi.org/10.1007/978-3-031-20047-2_1

[16] J. Shang, A. A. Zhang, Z. Dong, H. Zhang, and A. He, "Automated Pavement Detection and Artificial Intelligence Pavement Image Data Processing Technology," *Automation in Construction*, vol. 168, article 105797, 2024. https://doi.org/10.1016/j.autcon.2024.105797

[17] L. Zheng, J. Xiao, Y. Wang, W. Wu, Z. Chen, D. Yuan, and W. Jiang, "Deep Learning-Based Intelligent Detection of Pavement Distress," *Automation in Construction*, vol. 168, article 105772, 2024. https://doi.org/10.1016/j.autcon.2024.105772

[18] H. Gong, L. Liu, H. Liang, Y. Zhou, and L. Cong, "A State-of-the-Art Survey of Deep Learning Models for Automated Pavement Crack Segmentation," *International Journal of Transportation Science and Technology*, vol. 13, pp. 44–57, 2024. https://doi.org/10.1016/j.ijtst.2023.11.005

[19] G. Guo and Z. Zhang, "Road damage detection algorithm for improved YOLOv5," *Scientific Reports*, vol. 12, article 15523, 2022. https://doi.org/10.1038/s41598-022-19674-8

[20] K. Guo, C. He, M. Yang, and S. Wang, "A pavement distresses identification method optimized for YOLOv5s," *Scientific Reports*, vol. 12, article 3542, 2022. https://doi.org/10.1038/s41598-022-07527-3

[21] S. Xiao, K. Shang, K. Lin, Q. Wu, H. Gu, and Z. Zhang, "Pavement crack detection with hybrid-window attentive vision transformers," *International Journal of Applied Earth Observation and Geoinformation*, vol. 116, article 103172, 2023. https://doi.org/10.1016/j.jag.2022.103172

[22] F. Guo, J. Liu, C. Lv, and H. Yu, "A novel transformer-based network with attention mechanism for automatic pavement crack detection," *Construction and Building Materials*, vol. 391, article 131852, 2023. https://doi.org/10.1016/j.conbuildmat.2023.131852

[23] Y. Jiang, "Road damage detection and classification using deep neural networks," *Discover Applied Sciences*, vol. 6, article 421, 2024. https://doi.org/10.1007/s42452-024-06129-0

[24] S. Wang, B. Cai, W. Wang, Z. Li, W. Hu, B. Yan, and X. Liu, "Automated detection of pavement distress based on enhanced YOLOv8 and synthetic data with textured background modeling," *Transportation Geotechnics*, vol. 48, article 101304, 2024. https://doi.org/10.1016/j.trgeo.2024.101304

[25] H. Yao, Y. Fan, Y. Liu, D. Cao, N. Chen, T. Luo, J. Yang, X. Hu, J. Ji, and Z. You, "Development and optimization of object detection technology in pavement engineering: A literature review," *Journal of Road Engineering*, vol. 4, no. 2, pp. 163–188, 2024. https://doi.org/10.1016/j.jreng.2024.01.006

[26] S. Cano-Ortiz, L. Lloret Iglesias, P. Martinez Ruiz del Arbol, P. Lastra-Gonzalez, and D. Castro-Fresno, "An end-to-end computer vision system based on deep learning for pavement distress detection and quantification," *Construction and Building Materials*, vol. 416, article 135036, 2024. https://doi.org/10.1016/j.conbuildmat.2024.135036

[27] R. Khanam and M. Hussain, "YOLOv11: An Overview of the Key Architectural Enhancements," arXiv:2410.17725, 2024. https://arxiv.org/abs/2410.17725

Appendix A – Final Evaluation Provenance

The final four-arm comparison used the frozen public test manifest `experiments/temporal_filter_ablation_v2/public_test.txt`, containing 9,867 entries. Its SHA256 checksum was:

`3ef799db69b0e231be1fc872ecd6dcb2b0496a22ea3b22a2a6cfcae26cbd37c9`

The final standalone test evaluations were:

- **Control:** SLURM job `11672661`; best-checkpoint SHA256 `f8d900c162bf39f9755c50318a01f70b4975ec8d51fd5d7f049a41b6cbbe24ca`.
- **Filtered/subsampled:** SLURM job `11672662`; best-checkpoint SHA256 `6be956757e412457814d17f402a3f555bae146fa9814b292b71b0c9235c48221`.
- **Unfiltered:** SLURM job `11672663`; best-checkpoint SHA256 `4924d898debc314d193ba3a2c04d347372e7ba07288aa2047b32204d13079b09`.

The three original-condition evaluations completed successfully with exit code `0:0`. The additional equal-size Random1060 evaluation completed under SLURM job `12448051` and is reported in Chapter 5. Its checkpoint checksum should be appended to the provenance record only after verification against the original Beocat artifact. The selected checkpoints corresponded to validation-selected epochs 93 (control), 91 (filtered/subsampled), and 92 (unfiltered). The test set was not used to select those checkpoints.

The corresponding final training jobs were `11535772` (control), `11535773` (filtered/subsampled), and `11535774` (unfiltered); each completed 100 epochs. The recorded project Git revision for the training environment was `ea0576339893b4df3ce7cdc9a2369a62ec129b47`. The final launcher scripts were untracked at that revision, so their content is identified separately by SHA256 rather than being claimed as part of the commit:

- control launcher: `ff51f9423fa41010cf06b3a1529efbb10e13c8efff59563ca3f00ba47a24e71c`;
- filtered launcher: `21caac99fee348c3ca152d3a0aa604cc973ca2943ffb9f29aa3c2f4b05175435`;
- unfiltered launcher: `569eee4a1e28a4d6bed9fabaace64fdc32a58ecb50bb55c30b3c12bb842110e4`.

Appendix B – Software Artifact and Reproducibility Guide

This appendix uses the same authoritative `user_manual.tex` distributed with the project repository and intended for operational handoff to KDOT. Maintaining one source avoids divergence between the thesis appendix, the GitHub documentation, and the manual supplied to project users. Standalone-only cover and contents pages are suppressed when the manual is included here; the operational instructions themselves are identical.

Document control. This file is the single authoritative manual source for the GitHub repository, KDOT software handoff, and thesis appendix. Operational changes to the pipeline should be accompanied by a version/date update in this file so those three copies remain aligned.

B.1 Quick Start — I Just Want to Run It

Technical users: this is all you need. For step-by-step instructions, continue with the sections below.

```
# 1. Install Python, Git, and Miniconda
# Windows -- run in PowerShell or Command Prompt:
winget install --id Python.Python.3.11 -e
winget install --id Git.Git -e
winget install --id Anaconda.Miniconda3 -e

# macOS -- run in Terminal:
brew install python@3.11 git miniconda

# 2. Clone to Desktop and enter the project folder
#    Windows first: cd %USERPROFILE%\Desktop
#    macOS first:   cd ~/Desktop
git clone https://github.com/DeekshithaKalluri/Pavement_Distress_Detection.git
cd Pavement_Distress_Detection

# 3. Create and activate the Python environment
conda create -n pdd_env -c conda-forge python=3.11 -y
conda activate pdd_env
pip install ultralytics opencv-python easyocr
pip install pandas numpy matplotlib Pillow huggingface_hub
```

```
# 4. Verify the environment is working
python -c "import ultralytics; print(ultralytics.__version__)"
python -c "import cv2; print(cv2.__version__)"

# 5. Download the pretrained model weights (run these two commands)
hf download Deeksha9/pavement-distress-detection yolov11x_best.pt --local-dir weights
python -c "import os; os.replace('weights/yolov11x_best.pt','weights/best.pt')"

# 6. Run the pipeline from the project root
# (you are already here if you followed Step 2)
python auto_label_pipeline.py --input your_video.mp4
```

What you should see after Step 6. A new `auto_labels/01_your_video.mp4_output/` folder will appear inside the project directory, containing:

- `images_annotated/` — frames with coloured bounding boxes
- `labels/` — YOLO `.txt` annotation files
- `manifest.json` and `metadata.csv`

If the folder appears in your home directory instead, you ran the command from the wrong location. See the pipeline-running section below for the fix. If the folder is empty or the script exits immediately, see the Troubleshooting section.

If any step fails, the detailed sections below explain why and how to fix it.

B.2 Minimum Viable Test

Before processing hours of footage, confirm the full pipeline works on a short clip.

1. Copy any short dashcam clip (10–30 seconds) into `input_videos/`.
2. From the project root, run:

```
python auto_label_pipeline.py --input input_videos
```

3. Confirm the output folder exists:

```
# Windows
dir auto_labels

# macOS
ls auto_labels/
```

4. Open one image from `auto_labels/01_*/images_annotated/` and check that bounding boxes appear.

5. Open `metadata.csv` in Excel or a text editor and confirm it has one row per detection.

If all five checks pass, your installation is working and you can proceed with your full footage batch. If anything fails, see the Troubleshooting section before running on large batches.

B.3 What This Project Does

This tool detects pavement distress in roadway imagery using a YOLO11-based object detector and automatically creates labels for new dashcam video or still-image collections.

For video input, the pipeline samples frames, performs YOLO detection and ByteTrack-based temporal association, splits tracker observations into contiguous runs, requires at least three tracked observations, and tolerates at most one missed sampled frame between successive observations. Qualifying runs are then reduced to first/middle/last representative frames. The pipeline can also read the on-screen GPS overlay and writes YOLO labels plus human-readable metadata. For still-image input, detection is performed without tracking or temporal-persistence filtering.

Output file roles.

- `images/` — Retained unannotated candidate frames selected for review/retraining.
- `images_annotated/` — Retained frames with bounding boxes drawn; use for human review and quality control.
- `labels/` — YOLO-format `.txt` annotation files; use for YOLO training.
- `manifest.json` — Full structured detection record; use for programmatic analysis.
- `metadata.csv` — Detection data in tabular form; open in Excel or Google Sheets.

Operational safeguards.

- Auto-generated boxes are review candidates; they are not automatically validated ground truth.
- A usable image with no target distress is a valid negative example and should keep an empty matching YOLO label file.
- When comparing retraining cycles, keep validation/test data fixed and add new reviewed footage to training only.
- OCR-derived GPS coordinates are approximate metadata, not survey-grade positioning.
- Do not infer physical crack length from bounding-box pixels without calibrated camera geometry.

B.4 System Requirements

Requirement	Minimum	Recommended
Operating system	Windows 10/11 or macOS 12+	Windows 11 or macOS 13+
RAM	8 GB	16 GB or more
Disk space	15 GB free	40 GB free for large batches
Python	3.10 or 3.11	3.11
GPU	Not required	NVIDIA GPU with 8 GB+ VRAM (Windows)
Internet	Required for initial setup	Stable broadband connection

Apple Silicon Macs (M1/M2/M3/M4) can use PyTorch's MPS backend for GPU acceleration. This manual uses the CPU-compatible setup for simplicity. See `https://pytorch.org/docs/stable/notes/mps.html` for MPS configuration details.

Before You Begin

Check each item before starting:

- ☐ I know where my project folder will be located on this machine.
- ☐ I have a sample dashcam video (`.mp4`) ready for testing.
- ☐ I have approximately 30 minutes of uninterrupted time.

- ☐ I understand that output is saved *relative to where I run the command* — run everything from inside `Pavement_Distress_Detection/`.

B.5 Installing Prerequisites

B.5.1 Windows

Open PowerShell or Command Prompt and run:

```
winget install --id Python.Python.3.11 -e
winget install --id Git.Git -e
winget install --id Anaconda.Miniconda3 -e
```

Each installer may show a User Account Control prompt — approve it and allow the installer to finish. When all three commands complete, close the terminal and open **Anaconda Prompt** (search for it in the Start menu). Use Anaconda Prompt for all remaining steps on Windows.

Verify the installations from Anaconda Prompt:

```
python --version
git --version
conda --version
```

If `winget` is not recognized, update *App Installer* from the Microsoft Store and reopen the terminal.

B.5.2 macOS

Install Homebrew first:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
    install.sh)"
```

Then install the prerequisites:

```
brew install python@3.11
brew install git
brew install miniconda
conda init zsh
```

Close and reopen Terminal, then verify the installations:

```
python3 --version
git --version
```

```
conda --version
```

B.6 Getting the Project

Clone the repository to your Desktop so the project folder is easy to find in File Explorer or Finder.

Windows (Anaconda Prompt):

```
cd C:\Users\%USERNAME%\Desktop
mkdir PavementDistress
cd PavementDistress
git clone https://github.com/DeekshithaKalluri/Pavement_Distress_Detection.git
cd Pavement_Distress_Detection
```

macOS (Terminal):

```
cd ~/Desktop
mkdir PavementDistress
cd PavementDistress
git clone https://github.com/DeekshithaKalluri/Pavement_Distress_Detection.git
cd Pavement_Distress_Detection
```

A GitHub account is not required. The repository can also be downloaded with the green **Code** button on the GitHub page and selecting **Download ZIP**.

Where does the clone go? Running `git clone` creates a `Pavement_Distress_Detection/` folder inside your current directory. Using Desktop as the starting point (as above) means you can always see and find the project folder in File Explorer or Finder. If you accidentally run commands from a different directory, `auto_labels/` will appear there instead — Desktop makes mistakes like this immediately visible.

B.7 Setting Up the Python Environment

Create the isolated environment:

```
conda create -n pdd_env -c conda-forge python=3.11 -y
```

Conda Terms of Service behavior can vary. A real setup recording showed that the command above still produced a `CondaToSNonInteractiveError` because the configured channel set included Anaconda default channels. If this occurs, accept the three channel terms and rerun the create command:

```
conda tos accept --override-channels --channel https://repo.anaconda.com/pkgs/main
conda tos accept --override-channels --channel https://repo.anaconda.com/pkgs/r
conda tos accept --override-channels --channel https://repo.anaconda.com/pkgs/
    msys2
conda create -n pdd_env -c conda-forge python=3.11 -y
```

Activate the environment and install dependencies:

```
conda activate pdd_env
pip install ultralytics opencv-python easyocr
pip install pandas numpy matplotlib Pillow huggingface_hub
```

Verify that the key packages installed correctly:

```
python -c "import ultralytics; print(ultralytics.__version__)"
python -c "import cv2; print(cv2.__version__)"
```

Both commands should print a version number with no error. If either fails, confirm `(pdd_env)` is shown at your prompt and rerun the `pip install` command above.

Upgrading

If a new version of the script or weights is released:

- **New weights file:** download `best.pt` again with the Hugging Face command from Section 9 and overwrite `weights/best.pt`. You do not need to rebuild the conda environment.
- **Updated script:** run `git pull` from the project root. If new Python packages were added, rerun `pip install -r requirements.txt` inside `pdd_env`.
- **Breaking change to the environment:** delete and recreate it:

```
conda deactivate
conda env remove -n pdd_env
conda create -n pdd_env -c conda-forge python=3.11 -y
conda activate pdd_env
pip install ultralytics opencv-python easyocr
```

```
pip install pandas numpy matplotlib Pillow huggingface_hub
```

B.8 Project Folder Structure

```
Pavement_Distress_Detection/       <- PROJECT ROOT (run all commands here)
|-- auto_label_pipeline.py         <- main script; run from here
|-- scripts/                       <- helper utilities; not run directly
|   |-- extract_frames.py          <- called internally by the pipeline
|   |-- track_and_filter.py        <- called internally by the pipeline
|   |-- generate_figures_tables.py <- generates report charts
|   |-- merge_cycle.py             <- merges auto-labels into dataset
|-- weights/                       <- put best.pt here after download
|   `-- best.pt
|-- input_videos/                  <- drop your dashcam footage here
|-- input_images/                  <- or drop still images here
|-- auto_labels/                   <- pipeline output lands here
|-- pics/                          <- generated charts and figures
`-- requirements.txt               <- pip dependencies list
```

The main script `auto_label_pipeline.py` is at the project root, not inside `scripts/`. The files inside `scripts/` are called internally by the pipeline; you do not run them directly except for `merge_cycle.py` and `generate_figures_tables.py`.

B.9 Downloading Pretrained Model Weights

Create the weights directory if needed, download the model file with the Hugging Face CLI, and rename it to the pipeline default `best.pt`:

```
mkdir weights
hf download Deeksha9/pavement-distress-detection yolov11x_best.pt --local-dir weights
python -c "import os; os.replace('weights/yolov11x_best.pt','weights/best.pt')"
```

You may see older versions of this command with `local_dir_use_symlinks=False`. That parameter is deprecated and ignored by current `huggingface_hub` versions — omit it. The two commands above provide a shorter, cross-platform alternative to the original recorded Python download command.

Do not rely on a direct `curl` download for the weights. A failed download may create an HTML page or Git LFS pointer only a few kilobytes in size. Confirm that `weights/best.pt` is a large binary model file (greater than 100 MB). A small file means the download failed and must be retried.

B.10 Running the Auto-Labeling Pipeline

Always run pipeline commands from the project root folder (the directory that contains `auto_label_pipeline.py`). If you run from your home directory or any other location, the `auto_labels/` output folder will be created there instead of inside the project.

Check your current directory before running:

```
# Windows
cd C:\PavementDistress\Pavement_Distress_Detection

# macOS / Linux
cd ~/PavementDistress/Pavement_Distress_Detection
```

If you need to run from a different location, append an absolute `--output` path to your pipeline command (the following are path arguments, not standalone commands):

```
# Windows example
--output C:\PavementDistress\Pavement_Distress_Detection\auto_labels

# macOS example
--output ~/PavementDistress/Pavement_Distress_Detection/auto_labels
```

The only required flag is `--input`. The script defaults to `--weights weights/best.pt` and `--output auto_labels`, so most runs need just one argument. Additional flags (`--fps`, `--conf`, `--device`, `--no-gps-ocr`) are documented in Section B.10.4.

B.10.1 Single video

Windows:

```
python auto_label_pipeline.py --input input_videos\my_drive.mp4
```

macOS:

```
python auto_label_pipeline.py --input input_videos/my_drive.mp4
```

`--weights` defaults to `weights/best.pt` and `--output` defaults to `auto_labels`, both relative to the project root. You only need to specify them if you want to use different paths.
Windows terminal commands accept either backslashes (`input_videos\my_drive.mp4`) or forward slashes (`input_videos/my_drive.mp4`) for `--input` paths. For portability, the YAML configuration examples in Section 13 use forward slashes.

Windows vs. YAML path style. In terminal commands (`winget`, `python`, `cd`), use backslashes: `input_videos\my_drive.mp4`. In YAML configuration files (for training), always use forward slashes on every platform: `path: C:/PavementDistress/dataset`. Forward slashes avoid escaping ambiguities in quoted YAML paths.

B.10.2 Folder of videos

```
python auto_label_pipeline.py --input input_videos
```

B.10.3 Folder of still images

```
python auto_label_pipeline.py --input input_images
```

For image folders, the pipeline skips frame extraction and temporal filtering.

B.10.4 Optional flags

Flag	Purpose	Default
`--weights PATH`	Model weights path	`weights/best.pt`
`--output PATH`	Base output directory	`auto_labels`
`--conf 0.35`	Minimum confidence threshold	0.25
`--fps 4`	Frames *extracted and saved to disk* per second of source video (e.g., `4` on a 30 fps video saves 1 in every 7–8 frames; intermediate frames are skipped entirely)	4
`--device auto`	Auto, CPU, or GPU index (e.g., `0`)	auto
`--no-gps-ocr`	Skip GPS text extraction entirely	off

Example with custom settings (using default weights, output directory, and device):

```
python auto_label_pipeline.py --input input_videos --conf 0.35 --fps 4
```

The pipeline reports frame extraction and detection progress during execution.

B.10.5 GPS OCR notes

The pipeline reads GPS coordinates from a visible text overlay burned into the video frame. It looks for a text strip at the bottom of the frame containing latitude and longitude numbers in the format used by Garmin dashcams (e.g., `39.1931 -96.5893 26 MPH` in white text on a dark background). Other dashcam brands that burn GPS data in a different colour, location, or format may not be recognised; in that case the GPS columns will be blank and you should pass `--no-gps-ocr` to avoid unnecessary processing time. To verify OCR is working, open `metadata.csv` and check whether the `latitude` and `longitude` columns are populated. Empty GPS columns are expected when the video has no overlay, when the overlay uses an unsupported font or layout, or when `--no-gps-ocr` is passed. Use `--no-gps-ocr` to skip this step and reduce processing time if GPS metadata is not needed.

B.11 Understanding the Output

Each processed video or image batch receives a numbered output folder. This folder is created *relative to your current working directory*, so always run from the project root (see the warning in Section B.10).

```
auto_labels/
```

```
‘-- 01_my_drive.mp4_output/
    |-- images/            <- retained unannotated candidate frames
    |-- images_annotated/  <- frames with boxes drawn (use for QA/human review)
    |-- labels/            <- YOLO .txt annotation files (use for training)
    |-- manifest.json      <- structured JSON (use for programmatic analysis)
    ‘-- metadata.csv       <- spreadsheet (open in Excel or Google Sheets)
```

Why fewer frames than expected. For each qualifying tracked distress run, the pipeline retains only three representative frames—the first, middle, and last observed frames—to avoid flooding the dataset with near-duplicate views. If a qualifying run spans many sampled frames, only those representative frames are written to the retained candidate set.

`manifest.json` stores one record per retained detection. A typical entry looks like this:

```
{
  "image": "GRME0006_f00079.jpg",
  "source_video": "GRME0006",
  "track_id": 5,
  "frame_index": 79,
  "distress_class": "Longitudinal Crack",
  "confidence": 0.847,
  "bbox_xyxy": [812, 543, 1232, 765],
  "bbox_diagonal_px": 471.2,
  "latitude": 39.1931,
  "longitude": -96.5893,
  "gps_ocr_confidence": 0.91,
  "gps_sign_corrected": false
}
```

`track_id` links the detection to a ByteTrack object identity; `frame_index` is the frame number within the source video; `bbox_xyxy` gives pixel coordinates as `[x1,y1,x2,y2]`; `gps_sign_corrected` flags whether the majority-sign correction was applied to fix an OCR sign-inversion error.

Figures B.1 and B.2 show real detections from `images_annotated/`.



Figure B.1: A real frame from `images_annotated/`, showing two Longitudinal Crack detections with confidence scores drawn on the dashcam frame. GPS overlay is visible at the bottom.



Figure B.2: A second real annotated roadway frame from the same Garmin video, showing two YOLO-predicted Longitudinal Crack regions with confidence scores of approximately 46% and 36%. One prediction overlaps a white roadway marking, illustrating why automatically generated labels require visual quality assurance.

Figure B.3 shows what `metadata.csv` looks like opened as a spreadsheet.

KDC_metadata_preview

image_name	latitude	longitude	gps_ocr_confidence	gps_sign_corrected	distress_class	confidence
GRMN0032_f00000.jpg	39.16799	-96.60373	0.892	FALSE	Longitudinal Crack	0.373569
GRMN0032_f00006.jpg	39.16806	-96.60375	0.859	FALSE	Longitudinal Crack	0.346876
GRMN0032_f00007.jpg	39.16806	-96.60375	0.909	FALSE	Longitudinal Crack	0.402093
GRMN0032_f00010.jpg	39.16811	-96.60377	0.824	FALSE	Longitudinal Crack	0.311734
GRMN0032_f00012.jpg	39.16811	-96.60377	0.848	FALSE	Longitudinal Crack	0.606354
GRMN0032_f00012.jpg	39.16811	-96.60377	0.848	FALSE	Longitudinal Crack	0.344362
GRMN0032_f00017.jpg	39.16817	-96.6038	0.731	FALSE	Longitudinal Crack	0.398985
GRMN0032_f00020.jpg	39.16824	-96.60384	0.845	FALSE	Transverse Crack	0.497194
GRMN0032_f00037.jpg	39.16859	-96.60391	0.876	FALSE	Pothole	0.701212
GRMN0032_f00042.jpg	39.16869	-96.60389	0.815	FALSE	Longitudinal Crack	0.319572
GRMN0032_f00044.jpg	39.16879	-96.60387	0.841	FALSE	Longitudinal Crack	0.719731
GRMN0032_f00046.jpg				FALSE	Longitudinal Crack	0.358116

Figure B.3: A real `metadata.csv` excerpt opened as a spreadsheet. The `latitude` and `longitude` columns are populated where the source video had a readable GPS overlay; rows with no GPS data have blank coordinate fields, which is expected.

Privacy note. GPS coordinates extracted by OCR may constitute personally identifiable or location-sensitive information in some jurisdictions. Ensure you have permission to process the video data and handle the resulting `metadata.csv` accordingly.

B.12 Running on GPU (Optional but Recommended)

The pipeline can run on CPU, but GPU acceleration is recommended for larger video batches and for model training. Actual speed depends on the GPU, software build, frame-extraction rate, OCR use, storage, and model size. The final thesis experiments used an NVIDIA RTX A4000; other hardware should be benchmarked locally rather than assumed to match those timings.

Step 1 — Check whether PyTorch can see your GPU

```
python -c "import torch; print(torch.cuda.is_available())"
```

If this prints `True`, check the GPU name with the following command and skip to Step 3:

```
python -c "import torch; print(torch.cuda.get_device_name(0))"
```

If it prints `False`, continue to Step 2. Do not run the GPU-name command when CUDA is unavailable.

Step 2 — Install the CUDA-enabled PyTorch build (Windows + NVIDIA only)

If this check is false on a Windows workstation that has an NVIDIA GPU, install a CUDA-enabled PyTorch build that matches the machine's driver using the official PyTorch installation selector:

`https://pytorch.org/get-started/locally/`

The exact install command changes as PyTorch and CUDA releases change, so this manual intentionally does not hard-code one CUDA wheel URL. Re-run the check from Step 1 after installation and confirm that it prints `True`.

Step 3 — Run the pipeline with GPU

Pass `--device auto` (the default) or force a specific GPU:

```
# Auto-detect GPU (default -- no flag needed):
python auto_label_pipeline.py --input input_videos

# Force GPU 0 explicitly:
python auto_label_pipeline.py --input input_videos --device 0

# Force CPU even if a GPU is present:
python auto_label_pipeline.py --input input_videos --device cpu
```

The terminal will print `[device] using GPU 0` or `[device] using CPU` at startup so you know which device is active.

B.13 Training or Retraining a Model

Most users who only need inference or auto-labeling should use the supplied pretrained weights. Retraining is appropriate when new reviewed data are available or when the operating domain differs materially from the existing training data. There is no universal minimum number of images or guaranteed training time: both depend on class balance, annotation quality, hardware, model scale, and domain shift. Start with reviewed data, preserve a fixed evaluation set, and measure whether the new model actually improves the intended use case.

A typical dataset layout is:

```
dataset/
|-- images/
|   |-- train/
```

```
|   `-- val/
`-- labels/
    |-- train/
    `-- val/
```

Example YAML configuration file:

```
# Use forward slashes in the path field on ALL platforms, including Windows.
path: C:/PavementDistress/dataset
train: images/train
val: images/val
nc: 6
names:
  0: Longitudinal Crack
  1: Transverse Crack
  2: Alligator Crack
  3: Pothole
  4: Patch
  5: Block Crack
```

YAML path style. Always use forward slashes (`/`) in the `path:` field, even on Windows. In double-quoted YAML paths, backslashes can introduce escape sequences; forward slashes avoid this pitfall.

For a basic training run (configure epochs and patience as needed):

```
yolo detect train data=your_dataset.yaml model=yolo11x.pt imgsz=896
```

To train on GPU, add the `device` flag (configure epochs and patience as needed):

```
yolo detect train data=your_dataset.yaml model=yolo11x.pt imgsz=896 device=0
```

B.14 Iterative Retraining Cycle

The pipeline can be used repeatedly as new roadway footage is collected. The recommended operational rule is simple: **auto-label new footage, review the retained candidates, add accepted data to training only, preserve a fixed validation/test set, retrain, and evaluate before deployment**. Do not assume that a later cycle is better merely because it contains more pseudo-labeled data.

Result established in the accompanying thesis. In the final controlled comparison, the filtered/subsampled workflow added 1,060 unique KDOT frames instead of 3,266 corrected unfiltered detector-positive frames—a 67.5% reduction in added KDOT training-frame volume—while the resulting models remained in a narrow performance range on the same frozen public test set. This is a data-efficiency result, not evidence that every retained label is correct or that accuracy necessarily improves after retraining.

One reproducible cycle, step by step

Step 1 — Run the pipeline on new footage

```
python auto_label_pipeline.py --input new_videos/ --weights weights/best.pt --output
    auto_labels
```

This produces numbered output folders containing retained images, annotated review copies, labels, and metadata.

Step 2 — Review and correct the auto-labels

Inspect `images_annotated/` for missed distress, false positives, and misclassified distress types. Each retained image has a corresponding YOLO-format label file.

- **Valid negative image:** keep the image and keep an *empty* matching `.txt` label file. Do not remove the label file merely because the image contains no target distress.
- **Incorrect box or class:** edit the matching label file manually or with a labeling tool such as LabelImg or CVAT.
- **Unusable image:** remove the image from the candidate dataset and remove its matching label file as the same QA action.

Treat auto-labels as candidates, not ground truth. Temporal selection reduces transient and redundant candidates but does not guarantee box-level correctness. Review is especially important before data are reused in later training cycles.

Step 3 — Create a new dataset version

```
python scripts/merge_cycle.py --existing dataset/ --new auto_labels/ --output
    dataset_cycle1/
```

Write each cycle to a new versioned directory so the previous dataset remains recoverable. Check image/label correspondence and filename collisions before training.

Step 4 — Preserve the evaluation partitions

For comparisons across cycles, **do not reshuffle the validation or test sets**. Add newly reviewed KDOT imagery to the training population only and reuse the same validation/test manifests. This is the protocol used by the final controlled experiment in the thesis and prevents new KDOT frames from leaking into evaluation.

Step 5 — Retrain under a recorded configuration

```
yolo detect train data=dataset_cycle1.yaml model=yolo11x.pt imgsz=896 device=0
```

For a controlled research comparison, initialize each condition from the same base pretrained weights and hold the training configuration fixed. An operational deployment may choose a different fine-tuning strategy, but that should be validated separately and should not be compared as if it were the same experiment.

Step 6 — Evaluate on the fixed validation/test data

```
yolo detect val model=runs/detect/train/weights/best.pt data=dataset_cycle1.yaml
```

Record the dataset version, checkpoint, software versions, and metrics. Do not promote a newly trained checkpoint simply because it is newer; promote it only after review and evaluation against the operational acceptance criteria.

Step 7 — Repeat with provenance

For each new batch, record the source videos, number of reviewed/accepted training frames, dataset version, model initialization, training settings, and checkpoint used for deployment. This prevents later cycles from becoming irreproducible.

B.15 Generating Report Figures and Tables

Windows:

```
python scripts\generate_figures_tables.py
```

macOS/Linux:

```
python scripts/generate_figures_tables.py
```

The script expects training output in a `runs/` directory and writes figures to `pics/`.

B.16 GPU vs. CPU Notes

See Section B.12 for GPU setup. Hardware performance varies substantially, so the manual does not prescribe universal CPU/GPU speedups or training-time estimates. For planning purposes, benchmark a short representative video and one small training run on the actual

workstation or cluster that will be used. Record the extraction rate, OCR setting, model, input size, device, and wall-clock time with the benchmark.

In the thesis environment, final training used an NVIDIA RTX A4000 with Ultralytics 8.4.41, Python 3.10.20, and PyTorch 2.11.0+cu130. Those details are reproducibility context, not a guaranteed performance specification for other systems.

B.17 Troubleshooting

`python` **not recognized (Windows)**	**Likely cause:** Python is not on `PATH`. **Fix:** use Anaconda Prompt, or reinstall Python with "Add to PATH" enabled.
`python` **not found (macOS)**	**Likely cause:** macOS uses `python3` outside Conda. **Fix:** use `python3`, or activate `pdd_env`.
`conda` **not recognized**	**Likely cause:** Miniconda is not initialized for the shell. **Fix:** open Anaconda Prompt on Windows or run `conda init zsh` on macOS.
Conda ToS error	**Likely cause:** default channels require acceptance of the Conda Terms of Service. **Fix:** run the three `conda tos accept` commands described in the environment-setup section.
Module import error	**Likely cause:** the environment is not activated. **Fix:** run `conda activate pdd_env`, then retry.
Zero detections	**Likely cause:** the confidence threshold is too high or the weights are corrupt. **Fix:** try `--conf 0.15` for diagnosis and confirm `weights/best.pt` is the expected model file.
Weights file only a few KB	**Likely cause:** an HTML page or Git LFS pointer was downloaded instead of the weight file. **Fix:** rerun the `huggingface_hub` download command in the pretrained-weights section.
`SyntaxError: invalid character`	**Likely cause:** smart quotes were introduced during PDF copy-paste. **Fix:** retype quotation marks manually or copy the command from the source file.
Output in wrong location	**Likely cause:** the command was run outside the project root. **Fix:** change into `Pavement_Distress_Detection/` first, or pass an absolute output path.

Path/file not found	**Likely cause:** a typo or incorrect path separator. **Fix:** on Windows, use **Copy as path** and pass the resulting path to `--input`.
Very slow processing	**Likely cause:** CPU-only execution. **Fix:** GPU execution is preferred; for diagnosis, lower the extraction rate or disable GPS OCR when location metadata are unnecessary.
Empty GPS fields	**Likely cause:** the source has no readable GPS overlay or `--no-gps-ocr` was used. **Fix:** this is expected behavior; YOLO detection labels are unaffected.

B.18 Command Quick Reference

```
# --- Environment ---
conda activate pdd_env

# --- Navigate to the project root FIRST ---
cd C:\PavementDistress\Pavement_Distress_Detection   # Windows
# cd ~/PavementDistress/Pavement_Distress_Detection  # macOS

# --- Run the pipeline ---
# Single video (Windows):
python auto_label_pipeline.py --input input_videos\my_drive.mp4

# Single video (macOS):
python auto_label_pipeline.py --input input_videos/my_drive.mp4

# Folder of videos:
python auto_label_pipeline.py --input input_videos

# Folder of still images:
python auto_label_pipeline.py --input input_images

# With custom settings:
python auto_label_pipeline.py --input input_videos --conf 0.35 --fps 4

# --- Other tools ---
# Verify environment:
python -c "import ultralytics; print(ultralytics.__version__)"
python -c "import cv2; print(cv2.__version__)"

# Generate report figures (requires a completed training run in runs/):
```

```
python scripts/generate_figures_tables.py

# Train or fine-tune a model:
yolo detect train data=your_dataset.yaml model=yolo11x.pt imgsz=896

# Check whether an NVIDIA GPU is visible to PyTorch:
python -c "import torch; print(torch.cuda.is_available())"
# If False, use the official PyTorch install selector for the machine's driver.
```

B.19 FAQ

Do I need a GitHub account? No. Downloading the ZIP does not require an account.

Can I use a laptop without a dedicated GPU? Yes. Processing is slower but fully supported on CPU.

Can I run this on Linux? Yes. Follow the macOS-style terminal commands.

Where are results saved? In the directory supplied to `--output`; the default is `auto_labels`, created relative to wherever you ran the command. Always run from the project root to keep output inside the project.

What video formats are supported? Common OpenCV-compatible formats including MP4, MOV, AVI, and MKV.

Will detection work without GPS text? Yes. Detection and labeling are unaffected; only the GPS columns in `metadata.csv` will be blank.

Can I use the auto-generated labels to retrain the model? Yes. Review the files in `labels/` with a labeling tool (such as LabelImg), correct any errors, then follow the iterative cycle in Section B.14.

How many cycles should I run? There is no fixed number. Run a new cycle when you have a meaningful batch of reviewed new data, and stop or revise the workflow when the fixed evaluation set no longer shows an operational benefit. More pseudo-labeled data do not guarantee better model quality.

Do I need a GPU? No. CPU execution is supported, but GPU acceleration is recommended for larger video batches and training. Benchmark the actual hardware rather than relying on a universal speed estimate. See Section B.12 for setup steps.

My `auto_labels/` folder appeared in my home directory instead of the project folder. How do I fix it? This happens when the pipeline is run from a directory other than the project root. `cd` into `Pavement_Distress_Detection/` first, then rerun. The old folder can be moved or archived manually.

B.20 Workflow Decision Tree

The diagram below shows the two main uses of the pipeline.

One-off labeling run
Collect footage → run pipeline → review `images_annotated/` → use `labels/` for downstream training or analysis

Iterative retraining with fixed evaluation data
Run pipeline on new footage
↓
Review `images_annotated/` → correct any wrong labels
↓
Merge into dataset (`merge_cycle.py`)
↓
Add accepted data to training only; preserve fixed validation/test sets
↓
Retrain from `yolo11x.pt` base weights
↓
Evaluate on the same fixed validation/test data
↓
If accepted: archive provenance and promote the validated checkpoint
↓
Repeat with next batch of footage

The Iterative Retraining Cycle section walks through the loop step by step with full commands. The Running on GPU section explains how to enable GPU acceleration to make each cycle faster.