AutoML CI/CD/CT Capstone Project Final Report

Sepehr Heydarian, Archer Liu, Elshaday Yoseph, Tien Nguyen

Table of contents

1 Executive Summary

2 Introduction

- **Motivation**
  - Wildfires are a growing environmental threat with devastating consequences.
  - According to Defence Research and Development Canada (DRDC), early detection is critical to minimizing damage and preventing spread.
- **Partner Context**
  - Our capstone partner, Bayes Studio, uses specialized cameras for environmental monitoring and wildfire detection.
  - These cameras act as edge devices — carrying a lightweight model for wildfire detection
- **Problem Statement**
  - The partner currently faces a manual and time-consuming workflow:
    - Labeling incoming environmental imagery
    - Fine-tuning a base detection model
    - Compressing models for edge deployment
  - This bottleneck limits their ability to rapidly adapt models to changing data.
- **Technical Requirements**
  - For effective deployment on edge devices, the model must:
    1. Be trained on high-quality labeled data
    2. Be fine-tuned to current wildfire imagery
    3. Undergo distillation and quantization to compress the model to meet hardware constraints
- **Our Solution**
  - We developed an automated, continuous, and dynamic machine learning pipeline that:
    - Performs semi-automated labeling to create high-quality data (images)
    - Fine-tunes a YOLOv8 object detection model on newly labeled data
    - Applies model compression techniques (distillation and quantization)
    - Outputs edge-deployable models for real-time wildfire detection

3 Data - Pipeline Input

- **Data Format**
  - The input to our pipeline consists of:
    - Images in standard formats (e.g., `.jpg`)
    - Corresponding YOLO-formatted `.txt` label files
      - Each line in a label file includes: class ID, normalized bounding box coordinates (x_center, y_center, width, height)
      - Example: `1 0.757593 0.34663 0.06101 0.3655`
- **Object Categories**
  - The detection task focuses on five classes:
    1. Fire
    2. Smoke
    3. Lightning
    4. Vehicle
    5. Person
- **Data Volume**
  - Partner expects approximately 500 new images per month
- **Role in Pipeline**
  - This incoming data serves as the foundation for the pipeline's three core stages:
    1. Labeling via dual-model process and human in the loop
    2. Data augmentation for increase diversity
    3. Base model re-training

4 Data Science Methods and Product

4.1 Automated Labeling and Matching (Outline)

1. **Motivation**
   - Manual annotation is time-consuming and error-prone.
   - Our dual-model strategy leverages the strengths of both YOLOv8 and Grounding DINO for pre-labeling.
   - The goal is to automatically generate reliable labels with minimal human input.
2. **Input**
   - Unlabeled images placed in `mock_io/data/raw/images/`

- Pre-trained detection models:
  - **YOLOv8** – trained specifically on our five target categories (Fire, Smoke, Lightning, Human, Vehicle)
  - **Grounding DINO** – accurate grounding with text prompts like "fire", "smoke", etc.

3. **Workflow Steps**

- Step 1: YOLOv8 runs on the images and outputs predicted bounding boxes and class labels.
- Step 2: Grounding DINO runs with category prompts to generate alternative object proposals.
- Step 3: A custom matching script compares both outputs using IOU thresholds to identify:
  - Agreed labels → moved to `mock_io/data/labeled/`
  - Disagreements or missing detections → moved to `mock_io/data/mismatched/pending/` for review
- Step 4: All matched results are saved in YOLO-style JSON files for downstream training and augmentation.

4. **Output**

- Automatically labeled dataset stored in `mock_io/data/labeled/`
- Uncertain or mismatched files routed to the human-in-the-loop review process
- Each label includes bounding box, class, and confidence score

5. **Impact**

- Significantly accelerates the annotation process
- Boosts label consistency and enables scalable dataset expansion

4.2 Human-in-the-loop (Outline)

1. **Motivation**

- A high-performing model requires a high-quality labeled dataset
- We aim to enrich this dataset by correcting mismatches rather than discarding them
- To boost data quality with minimal human effort, we introduce a human-in-the-loop workflow that streamlines the correction of labeling errors and avoids manual JSON editing.

2. **Input**

- JSON files with incorrect or uncertain predictions from YOLO
- Corresponding image files referenced in those predictions

3. **Workflow Steps**

   - Step 1: Place prediction files in `mock_io/data/mismatched/pending/`
   - Step 2: Run the script to convert them into Label Studio tasks using images from `mock_io/data/raw/images/`
   - Step 3: Review and fix labels visually in the intuitive Label Studio interface
   - Step 4: Export the reviewed results to `mock_io/data/mismatched/reviewed_results/`
   - Step 5: The script automatically updates the review status of each file for progress tracking

4. **Output**

   - Corrected label files (JSON), ready for training
   - Results are versioned automatically for easy tracking

5. **Impact**

   - Enrich good-quality training dataset with minimal manual effort

4.3 Augmentation (Outline)

1. **Motivation**

   - A single image can appear in many different forms under varying conditions (e.g., lighting, weather, occlusion, angle of camera).
   - Augmentation increases dataset diversity and robustness, helping the model generalize better to unseen scenarios.
   - Our partner specifically requested augmentations to be done before training, as in-training augmentation could be too computationally expensive and time-consuming.

2. **Input**

   - Labeled images and their corresponding YOLO-formatted annotation files
   - Images sourced from the matched outputs of the pre-labeling pipeline saved in `labeled` folder.
   - Augmentation parameters specified in `pipeline_config.json` (e.g., number of augmentations, probabilities, blur limits)

3. **Workflow Steps**

   - Step 1: Load each image and its corresponding annotation
   - Step 2: For each image, generate a fixed number of augmented versions (e.g., 3)
   - Step 3: Apply image transformations using the **Albumentations** library, including:
     - Horizontal flips
     - Brightness and contrast adjustment
     - Hue and saturation shifts

  - Gaussian blur (with a configurable limit)
  - Gaussian noise
  - Rotation
  - Color adjustment to grayscale
- Step 4: Update the bounding boxes to reflect each transformation - handled by Albumentations library automatically.
- Note that images that do not contain objects of interest will not undergo augmentation.
- Step 5: Save the augmented images and updated labels.

4. **Output**

- A significantly larger and more diverse labeled dataset ready for training including:
- Augmented images and annotations in json files
- Non-augmented images saved in `no_prediction_images` folder as they do not contain any of the object of interest.

5. **Impact**

- Reduces overfitting and improves model generalization across lighting conditions and camera angles
- Provides more training samples without requiring manual data collection or annotation

4.4 Training (Outline)

1. **Motivation**

- Once high-quality labeled and augmented data is available, the model must be re-trained to incorporate new information.
- Continuous re-training allows the detection model to adapt to changing wildfire patterns and newly collected data.
- Our goal is to automate the fine-tuning of a base YOLOv8 model using the most recent dataset, with minimal manual intervention.

2. **Input**

- Augmented labeled images and corresponding annotation files. The non-augmented images with no labels are also used as true negative.
- A pre-trained YOLOv8 base model, typically selected from:
  - The latest updated model in the model registry, or path to any compatible model (YOLO model) the user chooses
  - Or a fallback initial base model (e.g., `nano_trained_model.pt`)
- Training hyperparameters (e.g., epochs, learning rate, image size) are configured in `train_config.json`

  - Any training argument compatible with YOLOv8 settings can be defined in the `train_config.json`

3. **Workflow Steps**

   - Step 1: Load training configuration and locate the most recent base model
   - Step 2: Organize input data into train/val/test splits
   - Step 3: Fine-tune the base YOLOv8 model using the prepared data and configuration
     - Training is performed using the Ultralytics training API
     - Includes logging of model performance
   - Step 4: Save the updated model using a timestamped filename in the model registry folder
   - Step 5: Store metadata (e.g., training date, config, performance) in the model registry folder with same name as the updated model.

4. **Output**

   - A fine-tuned YOLOv8 model that incorporates the latest training data
   - Model file saved with a naming pattern like: `[YYYY-MM-DD]_[HH_MM]_updated_yolo.pt`
   - Corresponding metadata and training outputs stored alongside the model for tracking and reproducibility

5. **Impact**

   - Enables fine-tuning as new data becomes available monthly
   - Ensures the detection model stays current and improves over time
   - Modular structure allows partners to adjust hyperparameters or retrain older models as needed

4.5 Knowledge Distillation (Outline)

1. **Motivation**

   - Large models are computationally expensive for real-time wildfire detection
   - Knowledge distillation transfers expertise from a larger teacher model to a smaller student model
   - Enables faster inference while maintaining detection accuracy

2. **Input**

   - Teacher model: Pre-trained YOLOv8 model with proven wildfire detection capabilities
   - Student model: Smaller YOLOv8n architecture initialized with pretrained weights from COCO 80 classes.

- Training dataset with 5 classes (FireBSI, LightningBSI, PersonBSI, SmokeBSI, VehicleBSI)
- Configuration file specifying distillation parameters and hyperparameters

3. **Workflow Steps**

   - Step 1: Load and prepare both teacher and student models
   - Step 2: Freeze first 10 layers of student model (backbone) for stable feature extraction
   - Step 3: Train student model using combined loss function:
     - Detection loss (_detection = 1.0) for direct supervision
     - Distillation loss (_distillation = 2.0) for knowledge transfer:
       * Bounding box distillation (_dist_ciou = 1.0)
       * Classification distillation (_dist_kl = 2.0)
   - Step 4: Regular checkpointing and model state preservation
   - Step 5: Save final distilled model for downstream tasks.

4. **Output**

   - Distilled YOLOv8n model with reduced size and complexity
   - Training logs and metrics for performance analysis
   - Checkpoint files for training resumption
   - Final model saved in `mock_io/model_registry/distilled/latest/`

5. **Impact**

   - Reduced model size and computational requirements
   - Faster inference times suitable for edge deployment
   - Maintained detection accuracy through knowledge transfer
   - Enables efficient deployment on resource-constrained devices

4.6 Quantization (Outline)

1. **Motivation**

   - Even after model distillation, inference on edge devices (e.g., drones or cameras) can still be too slow or resource-intensive
   - Quantization reduces model size and speeds up inference by reducing the precision of model weights

2. **Input**

   - The distilled YOLO model from the previous distillation step of the pipeline
   - Optional: labeled images and JSON annotations for calibration (required for IMX quantization)

3. **Workflow Steps**
    - Step 1: Set quantization method in `quantize_config.json` (`"FP16"`, `"ONNX"`, or `"IMX"`)
    - Step 2: Based on method:
        - **FP16**: Converts weights to float16, fast and accurate, saved as `_fp16.pt`
        - **ONNX**: Converts to ONNX and applies dynamic quantization, saved as `_onnx_quantized.onnx`
        - **IMX**: Uses Ultralytics export with calibration data to create IMX-ready format (Linux only), saved as `_imx.onnx`
    - Step 3: Quantized model is saved in `mock_io/model_outputs/quantized/`
4. **Output**
    - A smaller, faster model ready for deployment:
        - ~50% size reduction with FP16, minimal accuracy drop
        - ~75% size reduction with ONNX, slight accuracy tradeoff
        - IMX-ready model for Sony Raspberry Pi AI Camera
5. **Impact**
    - The quantized model can run directly on edge devices, enabling real-time fire alerts in the field

5 Conclusions and Recommendations

5.1 Problem Recap

- Bayes Studio needed a continuous pipeline to keep ML models accurate and up to date
- Manual labeling, training, and deployment slowed down real-time wildfire detection and response

5.2 What We Delivered

- An **automated**, **continuous**, **dynamic**, and **efficient** pipeline that streamlines pre-labeling, human-in-the-loop review, model training, optimization, and deployment:
    - **Automated**: From labeling to training and compression, everything flows with minimal manual effort
    - **Continuous**: Updated models help label data in future runs, improving over time
    - **Dynamic**: Easy to plug in new models or modules as requirements change
    - **Efficient**: Greatly reduces model update time - from weeks to just hours

5.3 Limitations & Recommendations

Limitation	Recommendation
Local-only pipeline	Add cloud storage and remote triggering for better scalability
Scattered data and model files	Integrate a lightweight database to centralize images, labels, and model metadata
No monitoring interface	Build a simple dashboard to track pipeline runs and model performance in real time

6 References