UNIVERSIDAD COMPLUTENSE MADRID

ntic master

Master Big Data, Data Science & Business Analytics

Constructeye AI: Development of an End-To-End AI-Powered Computer Vision System for Occupational Health and Safety in the Construction Industry

Author: Roger González Sánchez

Table of Contents

Abstract

As the construction industry continues to record some of the highest accident rates globally—resulting in significant financial impacts through penalties, insurance premiums, and operational downtime—the effectiveness of Occupational Health and Safety (OHS) is critical. However, traditional manual supervision often proves insufficient and lacks scalability.

To address this challenge, this Master's Thesis presents 'SmartSite Safety', an automated monitoring system powered by Artificial Intelligence. Leveraging state-of-the-art object detection algorithms (YOLO) on Cloud infrastructure, the system identifies Personal Protective Equipment (PPE) compliance in real-time. The methodology encompasses a comprehensive Exploratory Data Analysis (EDA), Data Augmentation strategies to address class imbalance, and advanced feature engineering to derive risk metrics. The final outcome is a robust model deployed via an interactive web interface (Streamlit) along with REST APIs deployment in local and cloud environemnts (AWS), enabling a strategic transition from reactive to proactive safety management.

1. Introduction and Business Justification

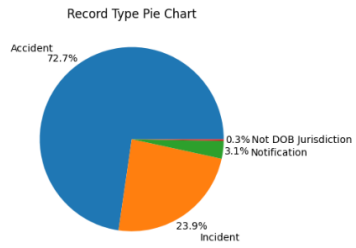
The construction field has the highest number of fatal and non-fatal accidents in the United States. From 2003 to 2010, more than 2,000 workers lost their lives because of traumatic brain injuries (TBI) (Trout et al., 2022).

The following data has been extracted from the New York City Department of Buildings, specifically from the "Construction-related incidents" dataset and displays accidents and injuries since January 2024 in New York City. This dataset consists of 1,000 records across various locations and buildings, featuring 20 different variables where I have focused primarily on the following variables:

- **record_type_description**: Incident category (accident, incident, notification, Not DOB Jurisdiction).
- **check2_description**: category of the cause associated with the incident (worker fell, other construction related, material failure, mechanical construction equipment, scaffold/shoring installations, excavation/soil work).
- **fatality**: Whether it resulted in a fatality.
- **injury**: Whether it resulted in a injury.

The following tables displays type of record with its respective count and fataly incidents, and the category of the cause associated with the incident in relation to fatalities, calculating the fatality_rate. It is observed that this ratio was higher in the categories of excavation/soil work, mechanical construction equipment, and worker fell, all exceeding 2%.

	record_type_description	count	count_normalized	fatality
0	Accident	727	0.727	14
1	Incident	239	0.239	0
2	Notification	31	0.031	0
3	Not DOB Jurisdiction	3	0.003	0



fatality	0	1	fatality_rate
check2_description			
Excavation/Soil Work	34	1	2.86
Mechanical Construction Equipment	70	2	2.78
Worker Fell	381	8	2.06
Material Failure (Fell)	101	1	0.98
Other Construction Related	355	2	0.56
Scaffold/Shoring Installations	45	0	0.00

Consequently, this demonstrates that implementing safety measures is fundamental to addressing the problem of accidents in the construction sector and preventing occupational hazards. Thanks to the increased power of graphics cards in recent years, there has been a boom in various models based on Deep Learning, especially in Computer Vision. This is where YOLO (You Only Look Once) models come in; these are trained to detect and locate objects quickly and accurately in images and videos. Their combination of efficiency and performance makes them particularly suitable for real-time applications in environments like construction, where identifying risks and monitoring safety compliance automatically is essential. Furthermore, the availability of large volumes of visual data and the development of frameworks that allow these models to run directly on local devices (edge computing) has facilitated their practical implementation, democratizing the use of computer vision and expanding its applicability across different sectors.

For this reason, this project proposes the development of an end-to-end computer vision system based on Artificial Intelligence to diagnose the PPE (Personal Protective Equipment) of each worker. The implementation of this system could reduce accidents, improve regulatory compliance, and generate significant savings in safety and insurance costs.

You can visit the repository with the code and further analysis **here**.

2. Methodology and Technological Architecture

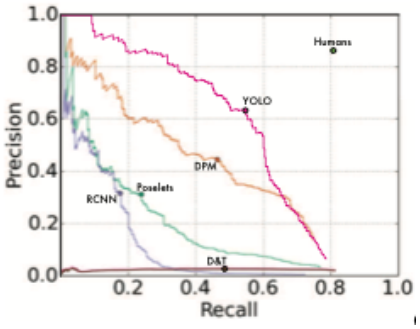
Roboflow is a development platform specifically designed for Computer Vision projects. Its main objective is to simplify the entire lifecycle of an AI model: from raw images to a functioning model deployed in an application or device. For the object detection process, this platform provides a very visual and simple way to draw and classify bounding boxes within images. However, for the development of this project, I have decided to use a pre-labeled dataset from Roboflow.

Objectives and Success Metrics

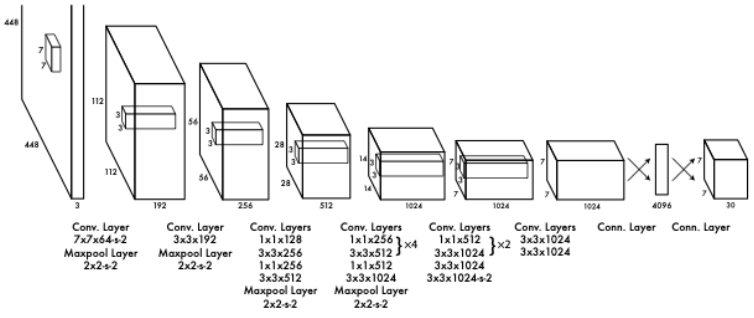
According to Goodfellow, Bengio, and Courville (2016), it is crucial to determine the model's objectives and which metrics to use for model selection. In the context of PPE detection, a high accuracy is not enough, as it can be misleading in imbalanced datasets. It is imperative to evaluate the model using Precision (to minimize false positives) and especially Recall, as both metrics are critical in industrial safety depending on the nature of the labels. In this context, maintaining a balance between error types is critical: **minimizing False Positives in PPE-present categories is just as vital as minimizing False Negatives in non-PPE categories**. Both scenarios represent a significant safety risk: the former creates a false sense of security by incorrectly identifying protective gear, while the latter fails to detect a direct violation representing both a fatality scenario.

Custom CNN from Scratch vs. Transfer Learning with YOLO

For the system implementation, developing a Convolutional Neural Network (CNN) architecture from scratch was ruled out due to the high demand for computational resources and time constraints. Instead, a Transfer Learning and Fine-Tuning strategy will be applied using the open-source YOLO architecture developed by Ultralytics. These architectures are known for their high performance, versatility, and improvements in precision and speed across various computer vision tasks. The choice of the YOLO architecture is based on its status as the industry standard for real-time object detection. The ability of these models to run in real-time applications and on-edge computing devices with minimal latency is the determining factor for their application in critical environments like construction (Ultralytics, 2025).

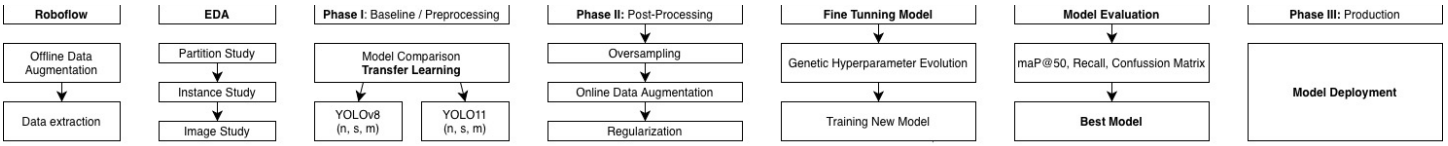


The structure of the YOLO architecture is presented below. The YOLO architecture employs a single convolutional neural network that processes the entire image in a single forward pass, dividing it into an S×S grid. “Unlike two-stage detectors that perform multiple iterations per image, such as those described in the illustration above, the YOLO architecture proposes an end-to-end approach where a single neural network simultaneously predicts bounding boxes and class probabilities" (Redmon et al., 2016). For this project, this architecture is leveraged through Transfer Learning, starting from a pre-trained model that has evolved from the initial 2015 version to current versions that optimize real-time detection.



Methodolody

I hereby present a diagram showing the different phases of this work's pipeline.



The process begins by loading the dataset through the Roboflow platform, accompanied by offline image processing with data augmentation to increase the dataset size (including rotations between -15° and 15°). This is followed by an exploratory data analysis (EDA) to study not only the different dataset partitions but also to conduct an analysis of instances and images. In the next phase, I have designed an experiment divided into two fundamental stages to determine the most efficient architecture for the proposed industrial safety environment.

In the first stage (Baseline), a comparative study between YOLOv8 and YOLOv11 will be conducted, evaluating the Nano, Small, and Medium sub-models in their original state, without applying data augmentation techniques. The objective of this phase is to select the model that offers the best trade-off between architectural complexity, computational cost, and critical metrics for ensuring the detection of all workers in at-risk situations.

In the second stage (Optimization), once the model with the best initial performance is selected, Data Augmentation and Oversampling techniques will be applied, specifically aimed at mitigating the minority class bias identified during the exploratory analysis. Following this, the **tuning process** will be carried out. Finally, I will carry out the deployment stage by different APIs development.

In the following section, I will conduct an exploratory study of the data prior to the baseline phase.

Here is my the notebook which can be followed (click here)

3. Data Acquisition and Exploratory Data Analysis (EDA)

The original dataset consists of 2,965 images in the training set, 119 in validation, and 90 images in the test set, with multiple instances per image. Across the entire dataset, a clear class imbalance is observed, with "person" and "helmet" being the most predominant classes, accounting for 70% of the data. However, "vest," "no-vest," and "no-helmet" make up the remaining 30%. This suggests that the model will initially be proficient at diagnosing high-incidence classes but will likely generate a significant number of false negatives.

	class_id	class_name	count	count_norm
0	3	person	7489	0.37
1	0	helmet	6734	0.33
2	4	vest	3481	0.17
3	2	no-vest	2347	0.12
4	1	no-helmet	311	0.02

Training Partition Analysis

I have created a dataframe "df" (see appendix) to extract various metrics. The columns are:

	image_name	class	x	y	w	h	area_pct	instances_per_img
0	ppe_1325_jpg.rf.3b698442655927821cd6b570a4fbf03a	helmet	0.726953	0.497222	0.056250	0.083333	0.468750	9
1	ppe_1325_jpg.rf.3b698442655927821cd6b570a4fbf03a	helmet	0.549609	0.468056	0.080469	0.116667	0.938802	9
2	ppe_1325_jpg.rf.3b698442655927821cd6b570a4fbf03a	helmet	0.805859	0.641667	0.088281	0.119444	1.054470	9
3	ppe_1325_jpg.rf.3b698442655927821cd6b570a4fbf03a	vest	0.718359	0.634028	0.107031	0.190278	2.036567	9
4	ppe_1325_jpg.rf.3b698442655927821cd6b570a4fbf03a	person	0.701172	0.721528	0.177344	0.556944	9.877062	9
...	...	...	...	...	...	...	...	...

- **class**: The category name, used to identify class imbalance.
- **x & y**: Normalized center coordinates of the object (0 to 1). These help detect whether objects are consistently centered or if the dataset provides spatial variety.
- **w & h**: Relative width and height. They indicate whether objects are predominantly elongated, tall, or square.
- **area_pct**: Calculated as (w×h)×100. This is vital for identifying small objects. If the area is below 0.5, the model will struggle to detect them without specialized techniques.
- **instances_per_img**: The number of objects in the image from which that row originates. It indicates whether scenes are simple (1 object) or congested (many objects), which influences parameters such as **Non-Max Suppression (NMS).**

Instance Study

First, I conducted an instance analysis to evaluate the class imbalance and the morphology of the bounding boxes. The following charts represent:

- **Instance Count per Class**: Displays the imbalance in "vest," "no-vest," and "no-helmet" instances.
- **Location Heatmap (Centers)**: Shows the density of object centers within the images. This suggests most workers were captured at a similar distance from the camera. The low density at the top and bottom edges indicates a lack of objects at extreme high-angle or low-angle shots. Consequently, the model may struggle to detect objects near the edges.
- **Relative Size per Class (%) on a Logarithmic Scale**: Shows the distribution of the area each class occupies relative to the total image. There are many records with very large areas, especially for "person." The fact that helmets are small objects justifies the use of architectures like YOLO, which include specific layers for small-scale object detection.
- **Objects per Image**: Displays the histogram of instances in each photograph. Most images contain fewer than 10 objects, but there is a "long tail" reaching up to 40 instances.
- **Width vs. Height (Shape)**: A scatter plot relating the width (w) and height (h) of each bounding box. Most points follow a vertical trend (height > width), which is consistent with human morphology.
- **General Size Distribution**: Shows many very small instances, which may cause the model to fail when observing distant subjects. Many objects occupy less than 10% of the image.

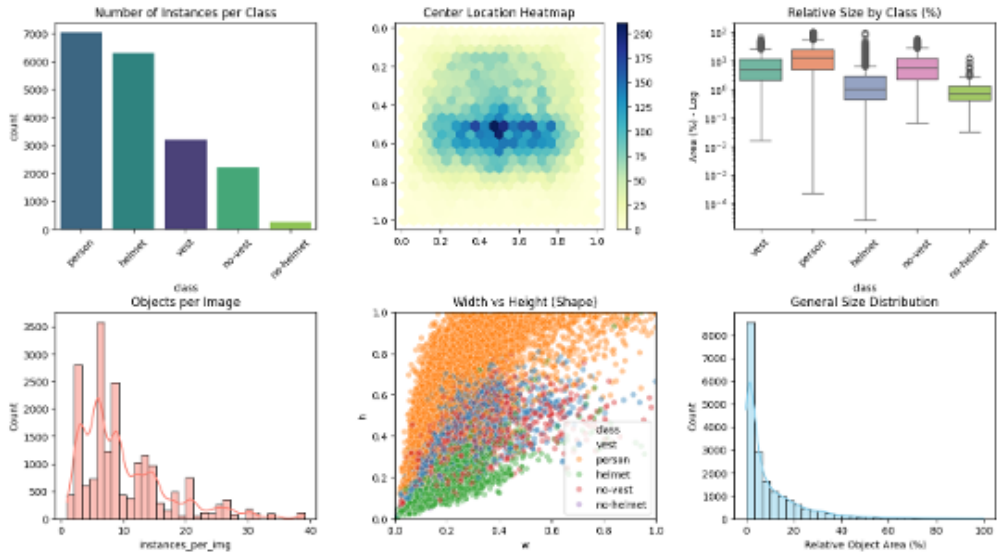


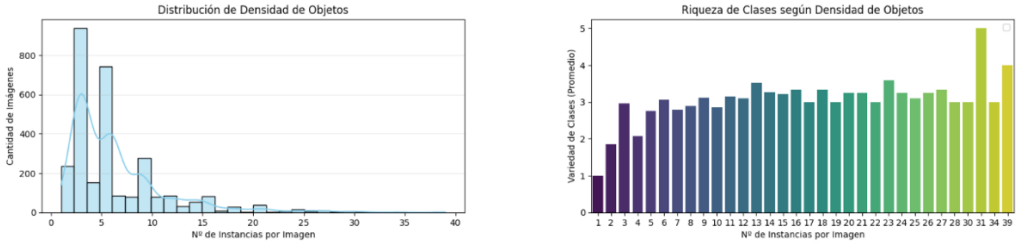
Image Study

To facilitate data comprehension, I derived a new dataset ("df_instances_per_img") calculating the number of unique classes and the number of instances per image for the training partition. Subsequently, I created "df_density," calculating the number of images and the mean diversity (average number of unique classes per instance count) for each instance level. The first 5 records for each dataset are shown below:

	image_name	class	instances_per_img
0	ppe_0604_jpg.rf.dcd50aa024e76d261e4ae44536f154dd	4	39
1	ppe_0604_jpg.rf.fada3f91c5648270c83457b605c0b755	4	39
2	ppe_0604_jpg.rf.fe923e56ca7c0dd4236d5e2d7ed1edd6	4	39
3	ppe_0444_jpg.rf.4dcc3091adf3a2768f0f147a0db5f255	3	34
4	ppe_0444_jpg.rf.8093f0251bdb4111a26fd273ea254016	3	34

	instances_per_img	total_images	diversidad_media
0	3	936	2.958333
1	6	597	3.068677
2	9	276	3.115942
3	2	205	1.858537
4	4	153	2.078431

The following charts were plotted to visualize these findings:



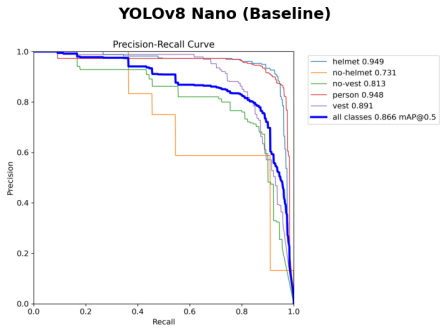
- **Instance Density Distribution**: A high concentration of data is identified in scenes of moderate complexity. Specifically, the dataset contains a significant volume of 936 images with 3 instances and 597 images with 6 instances. This representativeness is key, as it ensures the model is primarily trained on scenarios where the coexistence of multiple objects is the norm, simulating real industrial environments.
- **Instance Density Analysis**: By analyzing the average of unique classes based on the number of instances per image, two distinct phases are observed:

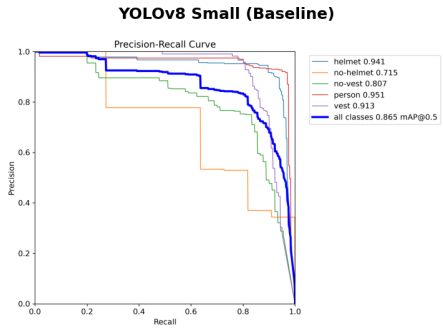
- **Optimal Progression Phase (1-3 instances):** In images with 1 to 3 objects, class variety increases almost linearly (reaching an average of 2.95 for 3 instances). This indicates that, in this range, practically every new object belongs to a different category.
- **Redundancy Phase (>4 instances):** Upon reaching 4 instances, a critical anomaly is detected where the average diversity drops to 2.07. This implies that, despite the increased visual load, there is a loss of taxonomic richness, with classes repeating across instances—meaning the dataset tends to repeat majority classes and omit...

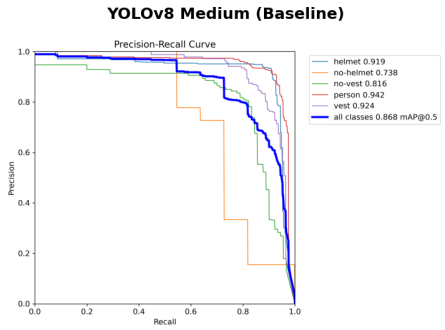
4. Phase I: Baseline. Model Comparison

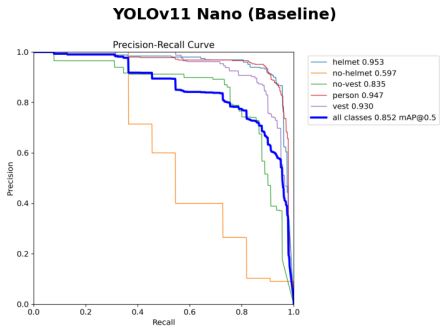
This section focuses on conducting a comparative study between different YOLOv8 and YOLOv11 models to determine which one performs best within the specific constraints of the project (**click here for the results**).

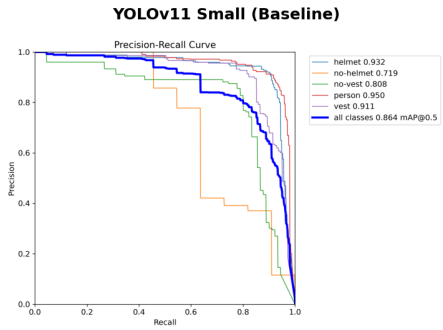
To ensure a fair comparison and prevent overfitting, I implemented an Early Stopping callback with a patience of 5 epochs. This technique allows each model to independently reach its optimal generalization point. Although the YOLO algorithm uses mAP@0.50 (mean Average Precision) as the default optimization metric during training, this study prioritizes Recall and Precision as the final selection metrics. To achieve this, I analyzed the Precision-Recall (PR) curves and the confusion matrices of each model, adjusting the Confidence Threshold to maximize the detection of risk situations, thereby minimizing the False Negative and Positive rates for critical PPE classes.

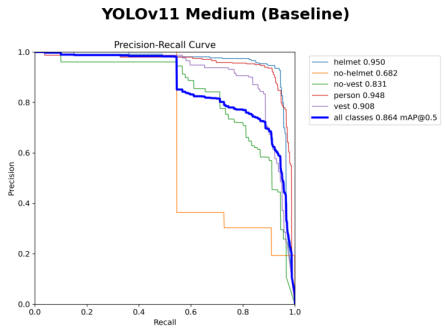












After analyzing the Precision-Recall curves from the Baseline phase, a significant degradation in performance is observed for minority classes, especially for "no-helmet." While "person" detection shows a reliability exceeding 94% across all

architectures, the "no-helmet" class drops to 59.7% in the YOLOv11n model. These results confirm the findings from the exploratory study: the imbalance of the original dataset limits the model's ability to identify critical situations.

Furthermore, due to the nature of the problem, I have decided to use the YOLOv8n model. Since it has fewer parameters (3.2 million vs. 25.9 million in YOLOv8m), it is a lighter model and significantly faster in terms of inference speed, making it suitable for mobile devices or IoT cameras. Although the YOLOv8m model could also be used—as it offers greater robustness and precision with a slightly higher recall—its increased layer depth would result in slower inference.

Therefore, the YOLOv8n model is selected for its efficiency to proceed to the Optimization phase through Data Augmentation, with the goal of improving the Recall curve for PPE non-compliance categories.

Once the optimal model was determined, the following metrics were obtained for the validation dataset using the best.pt file (the top-performing YOLOv8n model trained on the initial dataset). It is observed that the global recall reached 0.87, while the non-dominant classes, "no-helmet" and "no-vest," achieved recalls of 0.74 and 0.79, respectively. This indicates that 26% of workers without helmets were not detected by the system at this stage.

```
mAP@50: 0.8614893081981148
mAP50-95: 0.4858018708845892
Recall global: 0.8706768599612851
Recall de la clase helmet: 0.9481350381137659
Recall de la clase no-helmet: 0.7404277108433737
Recall de la clase no-vest: 0.7939664768736526
Recall de la clase person: 0.9481402914412564
Recall de la clase vest: 0.8767770237185255
```

While the standard YOLOv8n architecture consists of approximately 3.2 million parameters when configured for the COCO dataset (80 classes), the model implemented in this study has been streamlined to 3,006,623 parameters. This reduction is due to the adaptation of the output layer (head) to detect only the five specific industrial safety classes analyzed, thereby optimizing the system's computational efficiency.

5. Phase II: Processing. Data Augmentation and final model

Following the exploratory analysis of the dataset, a structural bias was identified in the relationship between instance density and class diversity. Specifically, it was observed that while images with 3 instances exhibit optimal label diversity (μ=2.95), in more complex scenes (4 instances), the variety drops drastically to an average of 2.07 (“image study” chart). This phenomenon indicates a redundancy of majority classes (person, hardhat...) and an under-representation of critical safety elements, such as the no-vest and no-hardhat classes (which account for only 17% of the total), as previously detailed in the "instance study" section.

	class_id	class_name	count	count_norm
0	3	person	7034	0.37
1	0	helmet	6307	0.33
2	4	vest	3211	0.17
3	2	no-vest	2196	0.12
4	1	no-helmet	276	0.01

Oversampling

The dataframe shows a significant disparity between the majority and minority classes. This imbalance typically causes the model to ignore minority classes in order to minimize the global Loss. To mitigate this bias, a Simple Random Oversampling strategy was implemented on the training set. This technique involved increasing the frequency of samples containing critical

instances. Specifically, a duplication factor of k=3 was applied to images featuring safety violations (no-hardhat, no-vest), while images showing regulatory compliance maintained a factor of k=1.

By tripling the presence of minority classes, the model's loss function is forced to assign greater weight to errors committed within these categories. This approach aims to improve Recall by reducing false negatives without the need to collect new data. The table below shows the new class distribution following the oversampling technique. Due to the high volume of "person" and "helmet" labels even in images containing "no-vest" and "no-helmet," the total count of dominant classes inevitably increased as well. However, in terms of percentage, no-vest now ranks as the third most frequent class, while vest has moved to fourth. In contrast, no-helmet remains the least frequent, as the original dataset contained so few records that significantly increasing its representation proved extremely challenging.

	class_id	class_name	count	count_norm
0	3	person	12942	0.36
1	0	helmet	11585	0.32
2	2	no-vest	6588	0.18
3	4	vest	4103	0.11
4	1	no-helmet	828	0.02

This technique can lead to model overfitting; therefore, it will be crucial to properly configure the model and enable specific data augmentation hyperparameters. To evaluate the model's performance, we will utilize mAP (mean Average Precision) per class and the F1-Score for the minority classes.

Data Augmentation and Regularization

It is important to highlight that, prior to training, the Roboflow platform was used to apply an "Offline" data augmentation stage (using random rotations between -15° and +15°) to increase the volume of samples from the original dataset.

Additionally, the native "Online" data augmentation parameters of the Ultralytics YOLO models have been integrated. This methodology allows random transformations to be applied at each epoch directly in the RAM, avoiding the saturation of the system's physical storage. Thanks to this "on-the-fly" processing, the model is exposed to subtly different variations in each iteration; for example, in a 50-epoch cycle, the network processes 50 distinct versions of the dataset, optimizing the learning of invariant features.

Specific geometric and chromatic techniques have been selected, tailored to the unique conditions of a construction site:

- **HSV Augmentation**: The model becomes more robust and capable of handling scenarios with varying lighting conditions, color schemes, and contrasts.
    - **HUE (hsv_h)**: Allows the model to learn to detect objects under different lighting conditions (simulating daylight or artificial light).
    - **Saturation (hsv_s):** Controls color vividness, providing the model with exposure to different color distributions.
    - **Value (hsv_v):** Affects image brightness, allowing the model to adapt to different light levels.
- **Degree Rotation (degrees):** By introducing rotations at a specific angle, the model becomes more robust to objects appearing in different orientations, such as when a worker tilts their head or bends over, changing the visual angle of the helmet.

- **Image Translation (translate):** This technique shifts the image horizontally or vertically within the frame. It is vital for simulating scenarios where workers are partially out of focus or entering/leaving the camera's field of view, teaching the model to recognize objects even when they are not centered.
- **Copy-Paste:** This technique leverages the instances from the oversampled dataset by pasting them onto different backgrounds. It is a critical mechanism for balancing the classes, as it increases the model's exposure to safety violations (minority classes) within diverse contexts.
- **Mixup**: This involves blending two training images to create a composite. It enhances the model's robustness against occlusions, which are frequent in crowded or cluttered construction environments.
- **Image Scale and Perspective (scale, perspective)**: These techniques involve rescaling and distorting the image to simulate changes in perspective and depth. They are fundamental in construction settings where safety elements appear at highly variable distances or from elevated viewpoints.
- **Mosaic Augmentation (mosaic):** This technique combines four training images into one, generating scenarios where the model is forced to differentiate categories in crowded environments with highly variable backgrounds.
- **Classification Loss Gain**: This forces the model to prioritize accurate categorization of critical safety equipment over localization, effectively reducing false negatives in the "no-helmet" and "no-vest" classes

```
augmentation_params = {
    "hsv_h": 0.015,      # Adjusts hue to simulate different lighting/weather conditions
    "hsv_s": 0.7,        # Controls saturation for color vividness variety
    "hsv_v": 0.4,        # Modifies brightness levels for light invariance
    "degrees": 10.0,     # Slight rotations to handle worker pose changes and head tilts
    "translate": 0.1,    # Shifts images to manage off-center framing or partial occlusions
    "scale": 0.6,        # Rescales objects to improve detection at varying distances
    "perspective": 0.001,# Compensates for distortions from elevated camera viewpoints
    "copy_paste": 0.6,   # High probability to leverage the oversampled "catalog" of rare classes
    "mixup": 0.15,       # Blends images to enhance robustness against site occlusions
    "mosaic": 1.0,       # Combines 4 images to train on high-density, cluttered environments
    "cls": 2.0           # Doubled weight for classification loss to prioritize critical PPE classes
}
```

The combined implementation of these techniques acts as a critical regularization mechanism. By diversifying exposure scenarios, the risk of overfitting is drastically reduced, preventing the model from memorizing noise within the training dataset. As a reinforcement for generalization, a Weight Decay (0.0005) has been configured to penalize excessive weights, along with an Early Stopping system with a patience of 8 epochs, ensuring that training ceases upon reaching the point of maximum predictive efficiency.

For illustrative purposes, the Albumentations Python library allows for the isolated replication and visualization of the transformations that YOLO executes internally during training (see Annexes)

Training

Once the augmentation parameters have been established, the training phase begins. It is important to note that while the model could have been trained at a resolution of 1280 pixels (which offers better detections for small or distant objects) a resolution of 640 pixels was chosen due to computational power constraints.

```python
# Model training
model.train(
    data=f'{dataset.location}/data.yaml', # Path to the configuration file defining class names and dataset paths.
    project=project,                      # Name of the main project directory to group related experiments.
    name=name,                            # Specific identifier for this run (creates a subfolder within the project).
    epochs=50,                            # Maximum number of complete passes through the training dataset.
    patience=15,                          # Early Stopping: halts training if no improvement is seen after 15 epochs.
    seed=42,                              # Sets a fixed random seed to ensure experimental reproducibility.
    imgsz=640,                            # Input image resolution; balances detection detail with computational cost.
    freeze=0,                             # Number of initial layers to freeze; 0 indicates full-network fine-tuning.
    box=5,                                # Weight gain for the bounding box regression loss in the total cost function.
    save_period=1,                        # Frequency (in epochs) at which model checkpoints are saved.
    lr0=0.001,                            # Initial learning rate for the optimizer's weight update steps.
    optimizer="AdamW",                    # Optimization algorithm with decoupled weight decay for better generalization.
    label_smoothing=0.1,                  # Regularization that prevents overconfidence by softening target labels.
    weight_decay=0.0005,                  # L2 penalty applied to weights to prevent growth and reduce overfitting.
    # Data augmentation:
    **augmentation_params                 # Unpacks and applies stochastic transforms (Mosaic, Mixup, etc.) to the pipeline.
)
```

During each epoch, the **model.train()** function provides several key metrics to monitor progress:

```
Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
 1/50      8.48G      1.713      7.153      1.706         88        640: 100% ━━━━━━━━━━━━ 338/338 2.5it/s 2:15
           Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% ━━━━━━━━━━━━ 4/4 1.5it/s 2.6s
             all        119        715      0.714      0.618      0.705      0.283

Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
 2/50      3.14G      1.596      5.349      1.601         36        640: 100% ━━━━━━━━━━━━ 338/338 2.7it/s 2:06
           Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% ━━━━━━━━━━━━ 4/4 2.4it/s 1.7s
             all        119        715      0.755      0.703      0.757      0.303
```

- **Box_loss**: Indicates the error in bounding box localization (how accurately the boxes fit the objects).
- **Cls_loss**: Measures the classification error (how correctly the model identifies the object's category).
- **Dfl_loss (Distribution Focal Loss):** Helps refine the box boundaries when they are not clearly defined or are ambiguous.
- **Box (P, R, mAP50, mAP50-95):** These are the Precision, Recall, and mAP metrics calculated on the validation set at the end of each epoch to monitor performance improvements.

By default, Ultralytics YOLO utilizes a metric called **Fitness** to determine whether the model is improving in each epoch and to trigger the Early Stopping callback. If the fitness score does not improve after 8 epochs, the callback is activated, and training stops. The system then saves the best.pt file containing the weights with the highest recorded fitness value (unless save_period is set to 1, which would save the .pt files for every epoch). This metric is a weighted sum of the following precision metrics:

Fitness = 0.1 * mAP50 + 0.9 * mAP50-90

6. Evaluation and metrics. Model Results

We will begin by analyzing the fundamental metrics that are not only crucial for training and evaluation but are also applicable to any object detection model within the YOLO family:

- **Intersection over Union (IoU):** Measures the overlap between the prediction and the ground truth; it is essential for tuning Non-Maximum Suppression (NMS) and validating localization accuracy.
- **Average Precision (AP)**: Summarizes model performance by combining precision and recall into a single value, calculated as the area under the PR curve
- **Mean Average Precision (mAP**): The average of AP values across all classes; it serves as the definitive global metric for evaluating multi-class detection models.
- **Precision and Recall**: measures accuracy (avoiding false positives), while Recall measures completeness (the ability to detect all real-world instances).
- **F1 Score**: The harmonic mean of precision and recall; it provides an ideal balance for evaluating performance when both metrics are equally critical.

Validation and Class-Specific Analysis: model.val()

Upon completing the training, model.val() is executed on the validation set. By default, Ultralytics sets a confidence threshold of 0.001 and an iou=0.70 to calculate the mAP, which allows for assessing the model's full capacity before applying aggressive filtering. This process breaks down performance by class, which is important for analyzing minority categories.

```
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95):
                   all        119        715      0.868      0.853      0.903      0.495
                helmet        117        232      0.899      0.948      0.958      0.549
             no-helmet          6         11      0.789      0.681      0.827      0.342
               no-vest         52         90       0.84      0.818      0.855      0.424
                person        115        241      0.927      0.954      0.956      0.634
                  vest         74        141      0.885      0.865      0.916      0.525
Speed: 6.8ms preprocess, 4.6ms inference, 0.0ms loss, 4.1ms postprocess per image
Results saved to /content/runs/detect/val2
```

Additionally, the model generates a directory containing various key performance metrics:

- **F1 Curve (F1_curve.png):** Represents the F1 Score across various confidence thresholds, helping to identify the optimal balance between false positives and false negatives.
- **Precision-Recall Curve (PR_curve.png):** A comprehensive visualization showing the relationship between precision and recall. It is particularly significant when working with imbalanced classes.
- **Confusion Matrix (confusion_matrix.png):** Displays a detailed count of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).
- **Normalized Confusion Matrix**: Represents the data in proportions, facilitating a direct comparison between classes of different sizes.
- **Validation Batches (Labels vs. Predictions):** Images that compare the ground truth labels with the model's predictions, allowing for a visual inspection of detection errors.

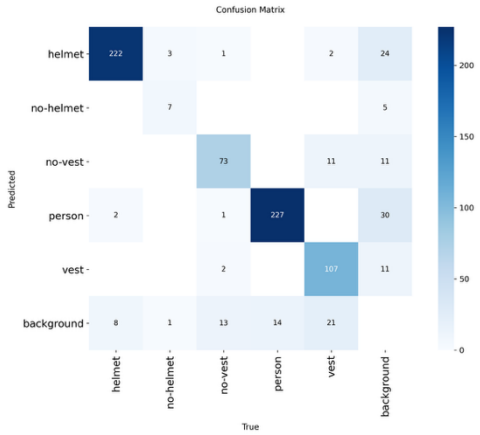
Ultimately, the evaluation strategy will prioritize minimizing false negatives in the "no-helmet" and "no-vest" classes, as well as false positives for the "helmet" and "vest" classes by adjusting the confidence and IoU parameters (which act as thresholds during the validation process). From a safety perspective, these errors represent the highest risks. Consequently, I have used Precision, Recall, and mAP@50 metrics to derive the following conclusions.

Class	Meaning of False Negative	Real Business/Safety Impact
Helmet	The worker is wearing a helmet, but the model does not detect it.	**False Alarm**: The system will trigger a violation alert when none has occurred.
No-helmet	The worker is NOT wearing a helmet, but the model does not detect it.	**Critical Risk**: A life-threatening violation goes unnoticed.
Class	**Meaning of False Positive**	**Real Business/Safety Impact**
Helmet	The worker is NOT wearing a helmet, but the model detects it as a helmet.	**Critical Risk**: The model "blindly" validates a dangerous situation as safe.
No-helmet	The worker is wearing a helmet, but the model predicts they are not.	**False Alarm**: Similar to the FN in the Helmet class.

Model 1: Baseline Model Results (Preprocessing Phase)

This model was previously trained using my dataset without any preprocessing or oversampling tasks (Phase I). This model was trained for 50 epochs with an early stopping mechanism set to a patience of 5. The best resulting model produced the following confusion matrix on the validation data (click here for the model directory):

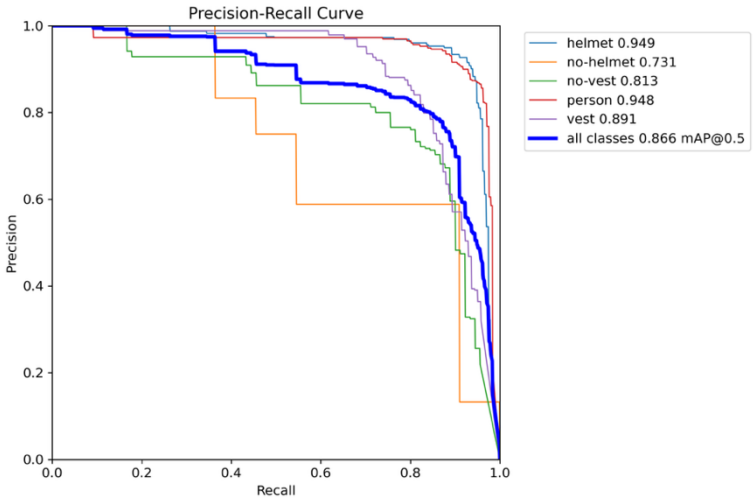
Class	*True*	*Predicted*	*True Positives*	*False Positives*	*False Negatives*	*Precision*	*Recall*
helmet	232	252	222	30	10	**0.881**	**0.956**
no-helmet	11	12	7	5	4	**0.583**	**0.636**
no-vest	90	95	73	22	17	**0.768**	**0.811**
person	241	260	227	33	14	**0.873**	**0.941**
vest	141	120	107	13	34	**0.892**	**0.759**

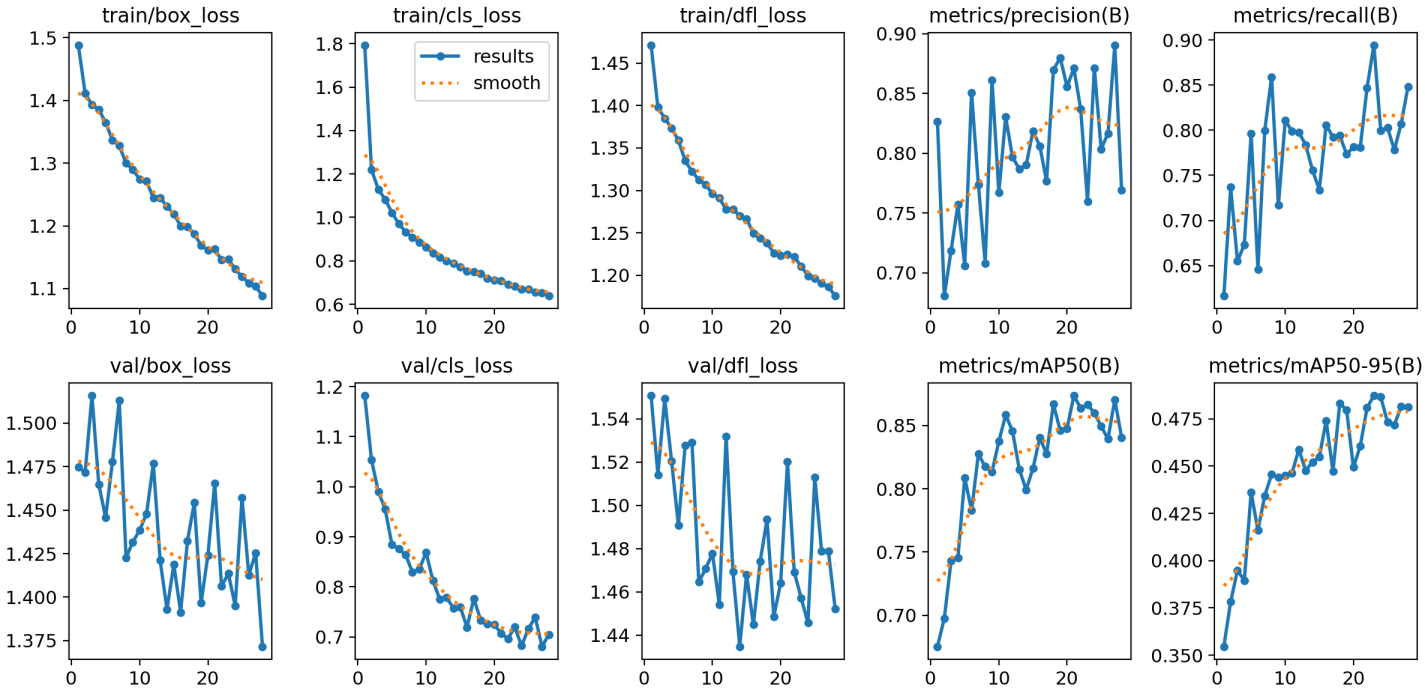


The confusion matrix demonstrates that the model is highly effective at identifying the person and helmet classes. However, the presence of numerous false negatives poses a significant challenge for safety applications. Specifically, the model failed to detect 21 vests, 14 people, and 13 no-vest instances. Furthermore, inter-class confusion is evident, as the model tends to misclassify no-vest as vest, which could lead to a dangerous false sense of security.

The following Precision-Recall Curve shows that the no-helmet class (0.731) "suffers" the most; it experiences a sharp drop in precision as recall increases, indicating the model struggles to distinguish unprotected heads amidst high scene variability.

Finally, the training progress is presented up to the activation of early stopping, a mechanism employed to prevent overfitting. The model achieved stable convergence; although a more permissive criterion could have been applied to this callback (given that the system automatically saves the weights of the best epoch) temporal efficiency was prioritized. As commented above, the primary objective of this stage was to establish a solid comparison between the YOLOv8 and YOLO11 architectures under controlled conditions.





Below, I'm showing an enhanced version of the model after oversampling, data augmentation and parameters as commented in the previous section.

Model 2: Results After Processing (Post-processing)

After generating new samples using oversampling techniques to balance the dataset, I proceeded with the model training using the following optimized parameters, in addition to the augmentation_params previously detailed. A comprehensive list of all hyperparameters is available in the args.yaml file generated by the system (click here for the model directory):

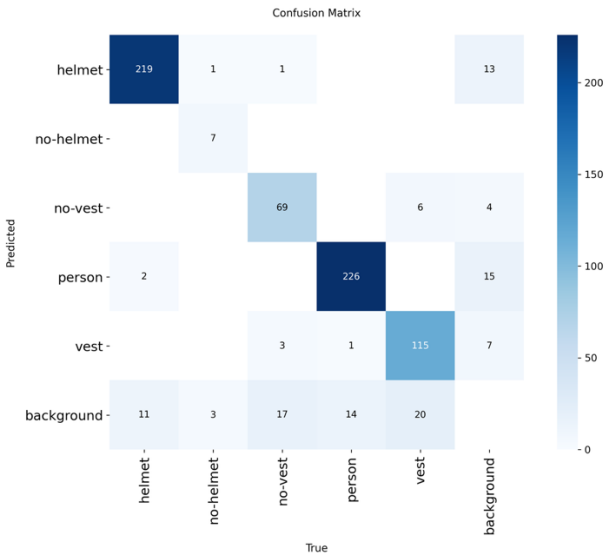
PARAMETER	VALUE	TECHNICAL OBJECTIVE
epochs	50	Sets the maximum number of complete training cycles to ensure the model sufficiently learns complex patterns in the construction site data.

patience	15	Implements **Early Stopping** by allowing 15 epochs without improvement before halting, ensuring the model reaches its full potential without wasting resources.
seed	42	Establishes a fixed reference for all stochastic processes, ensuring **experimental reproducibility** so that results can be verified in future runs.
imgsz	640	Defines the input resolution to provide a balance between the detection of small PPE items (like helmets) and the available computational power.
freeze	0	Specifies that no layers are frozen, allowing for **full-network fine-tuning** to adapt all weights specifically to the custom PPE dataset.
box	5	Increases the weight of the **bounding box loss** in the total cost function, prioritizing high-precision object localization and tighter fit around targets.
save_period	1	Ensures a model checkpoint is saved after every epoch, facilitating a granular audit of the training progress and performance peaks.
lr0	0.001	Sets the initial learning rate, determining the step size for weight updates to achieve a stable and efficient descent toward the global minimum loss.
optimizer	AdamW	Uses an advanced optimizer with decoupled **weight decay**, which is superior for generalizing patterns in deep architectures like YOLOv8 and YOLO11.
label_smoothing	0.1	Applies regularization by softening hard labels, preventing the model from becoming overconfident and improving its ability to handle noisy or ambiguous data.
weight_decay	0.0005	Implements L2 regularization to penalize large weights, effectively reducing the risk of **overfitting** and improving the model's robustness on unseen data.

The hyperparameters were selected to improve the performance metrics compared to the baseline model without significantly increasing training time. Validation was conducted using the 119 images from the 'val' partition of the dataset.

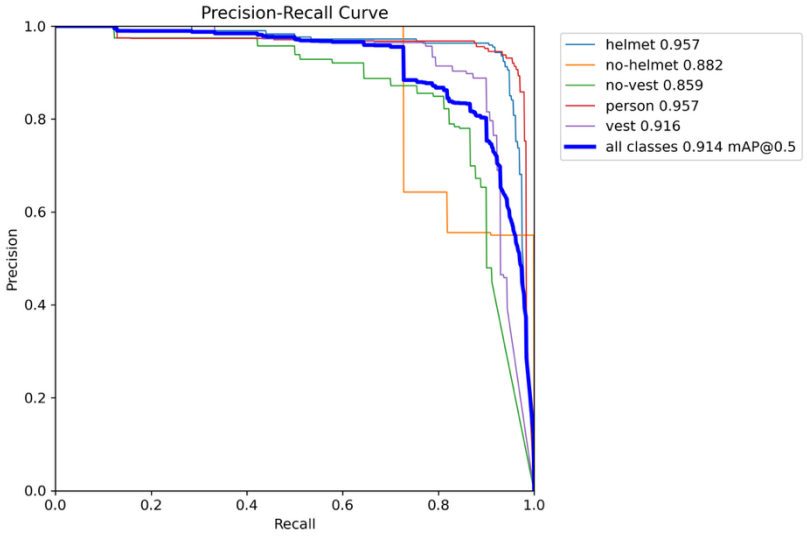
Class	*True*	*Predicted*	*True Positives*	*False Positives*	*False Negatives*	*Precision*	*Recall*
helmet	**232**	**234**	**219**	15	13	**0.936**	**0.944**
no-helmet	**11**	**7**	**7**	0	4	**1.000**	**0.636**
no-vest	**90**	**79**	**69**	10	21	**0.873**	**0.767**
person	**241**	**243**	**226**	17	15	**0.930**	**0.938**
vest	**141**	**126**	**115**	11	26	**0.913**	**0.816**

Precision has improved across all classes over recall. However, recall has decreased. Conversely, precision has significantly improved from 0.583 to 1 in the no-helmet class. This indicates that the model has stopped "hallucinating" violations; previously, nearly half of the "no-helmet" alerts were errors (likely misclassifying caps or hairstyles as a lack of protection). Now, when the model flags a missing helmet, there is an 100% probability that it is correct, drastically reducing the number of false positives and increasing system reliability.

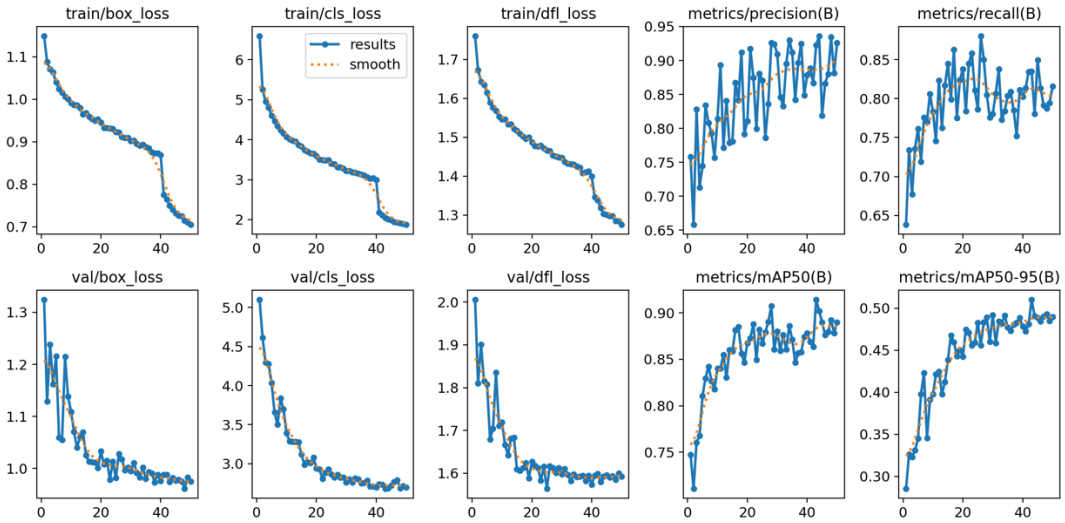


The fact that recall has not improved proportionally to precision confirms an important lack of samples. The model has learned the "indisputable" features of a person without a helmet, but due to limited exposure to more examples, it prefers to remain silent in ambiguous situations (e.g., poor lighting or the person's back to the camera). This creates a high number of False Negatives (FN).

Regarding the **Precision-Recall (PR) Curve**, this model outperformed the baseline, moving from 0.866 mAP@50 to 0.914 mAP@50. The PR curve demonstrates the trade-off between these metrics; for the "no-helmet" class (orange line), the staggered shape of the curve reinforces the need for more data. This line maintains high precision, but as soon as recall attempts to exceed 0.7, precision drops near 0.6. This confirms the model is only confident in detecting "no-helmet" instances under ideal conditions. When the scenario becomes challenging (low light, distance), the model either fails to predict (lower recall) or begins to misclassify (lower precision).



Finally, the training reached all 50 epochs because the patience was increased to 15, resulting in more stable validation curves (val/…). A notable performance jump is observed starting at epoch 40. This is due to the deactivation of Mosaic Augmentation in the final stages, allowing the model to "fine-tune" bounding box accuracy using clean, real-world images without the distortions generated by the mosaic process. By default, YOLO uses a close_mosaic=10 setting; with epochs=50, the model trains with aggressive augmentations until epoch 40. From epoch 41 onwards, it enters a stabilization phase to finalize the model weights.



While Mosaic Augmentation is excellent for helping the model detect small objects and various scales (improving Recall), it can introduce noise that prevents perfect coordinate precision toward the end of training. "Turning it off" in the final epochs allows the model to stabilize and refine its predictions.

Predictions (model.predict and model.track)

This phase is where I fine-tune various parameters such as confidence (conf), iou, and image_size (imgsz), which ultimately determine the system's real-world behavior in a construction environment. Since the main goal is to minimize fatal risks, the confidence threshold can be lowered (e.g., < 0.25). This adjustment ensures the system detects almost every potential instance, even though it may increase false positives in "no-PPE" classes and false negatives in "PPE" classes.

For video, **model.track()** replaces standard prediction by keeping unique IDs for each detectation. This ensures temporal consistency, allowing for a more robust analysis of PPE compliance as individuals move through the construction site.

Business logic for Video Analytics

To enhance the system's reliability in video processing, I have implemented a specialized business logic that defines head_region and torso_region by calculating their spatial intersections with the "person" bounding box. This approach was specifically designed to mitigate the limited number of records for minority classes, particularly the "no-helmet" category. By default, head_region is defined as the top 35% of the "person" bounding box whereas torso_region is defined between 20% and 80% of the vertical span of the "person" bounding box.

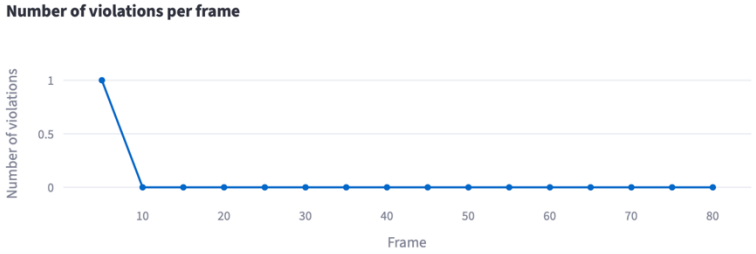
The system operates based on the following reasoning:

- **Person Detection**: The presence of a person is identified with high precision.
- **Equipment Verification**: Instead of analyzing the entire person's bounding box (bbox_person), the algorithm segments the body into functional areas. The following specific functions isolate the upper region (head) and the middle region (torso) by applying proportional ratios to the subject's total height:
    - **head_region**: Calculates the top boundary of the person, assuming the head area occupies approximately the upper 35% of the total detected height.
    - **torso_region**: Defines the area where a safety vest is expected, discarding the top 20% and bottom 20% to avoid overlap with the head or lower extremities.
    - **bbox_intersects**: A boolean verification algorithm that determines if a detected object is effectively located within the corresponding anatomical region.
- **Violation Logic**: If a person is detected but the "helmet" class is absent from the upper axis, or the "vest" class is missing from the middle section, the instance is automatically classified as a safety violation alert.

```python
def head_region(person_bbox, head_ratio=0.35):
    x1, y1, x2, y2 = person_bbox
    height = y2 - y1
    return (x1, y1, x2, y1 + head_ratio * height)

def torso_region(person_bbox, lower_part_ratio=0.2, higher_part_ratio=0.8):
    x1, y1, x2, y2 = person_bbox
    height = y2 - y1
    # El chaleco suele estar entre el 20% y el 80% de la altura
    return (x1, y1 + lower_part_ratio * height, x2, y1 + higher_part_ratio * height)

def bbox_intersects(b1, b2):
    xA = max(b1[0], b2[0])
    yA = max(b1[1], b2[1])
    xB = min(b1[2], b2[2])
    yB = min(b1[3], b2[3])
    return xA < xB and yA < yB
```

These functions are integrated into the **analyze_video_dataframe()** function, which also returns a transformed DataFrame based on the video's **frame_stride**. This variable was included because running the model on every single frame is computationally expensive and significantly slows down the process.

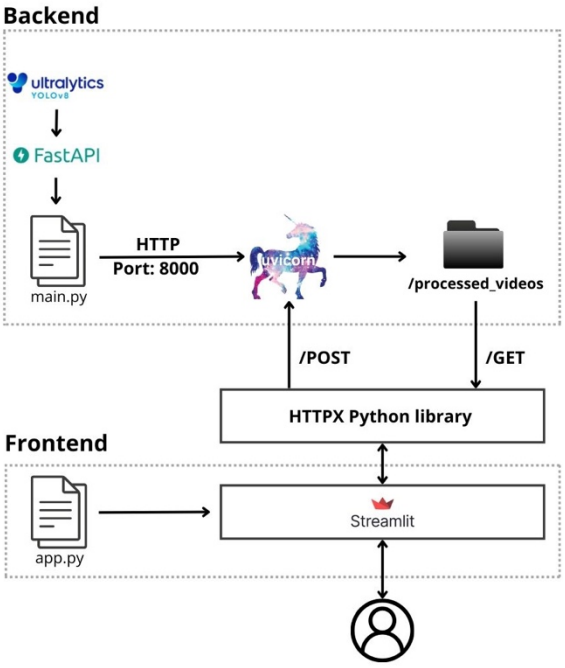
Thus, it is possible to analyze a video and determine frame-by-frame if any violation has occurred. For example, for a video with a frame_stride = 5, the following graph shows the detection of violations for each processed frame.



The result is exported as a Pandas DataFrame to allow for a detailed analysis of the different violations, the specific frame where they occurred, and the bounding boxes of all detected elements.

7. Deployment Ecosystem and Cloud Infrastructure

The deployment ecosystem is structured under a decoupled architecture, where the components interact through a REST API composed of several specialized endpoints (click here for code)



- **Backend (main.py): API Creation with FastAPI:** Developed with FastAPI and managed via the Uvicorn server, this component serves as the system's core. It handles the persistent loading of the model and manages the endpoints that process image and video requests:
    - **@app.get("/health"):** Verifies system status. It returns a plain JSON object and a 200-status code.
    - **@app.put("/prepare-output-dir")**: creates the output directory if it doesn't exist in a S3 bucket or locally, ensuring that the application can save processed videos without errors related to missing directories. It ensures the output directory exists for saving processed video.

- **@app.post("/image-specs"):** Extracts and returns metadata from the image uploaded by the user.
- **@app.post("/predict_image"):** Executes inference on an image and returns a combined response including the annotated image (encoded in Base64) and a DataFrame containing the detections. The use of Base64 allows for consolidating into a single request what would otherwise require multiple endpoints.
- **@app.post("/video-feed-df-specs"):** Provides metadata analysis for uploaded video files.
- **@app.post("/process-video_df"):** Processes an entire video and returns a JSON object containing the operation status, detections based on the proposed business logic, and the output filename.
- **@app.get("/download-video/{filename}"):** Facilitates the download of the processed video generated during the inference phase.
- **@app.get("/show-video/{video}"):** Enables direct visualization of the inferred video within the interface.
- **@app.post("/visualize-frame/{video_name}/{n_frame}"):** Allows for granular inspection of a specific frame within a video to analyze the model's detections at that precise moment.
- **@app.post("/predict-video_live"):** Manages real-time inference for live video streams.

- **Frontend (app.py): Streamlit Interface:** Implemented using Streamlit, it offers an intuitive dashboard that abstracts the technical complexity of the system. This interface allows the end-user to interact with the model seamlessly, facilitating real-time visualization of results and report management across three input modalities: Live Inference, images, and pre-recorded videos. This stage will be running in a local host which will connect to the backend through the HTTPX Python library.

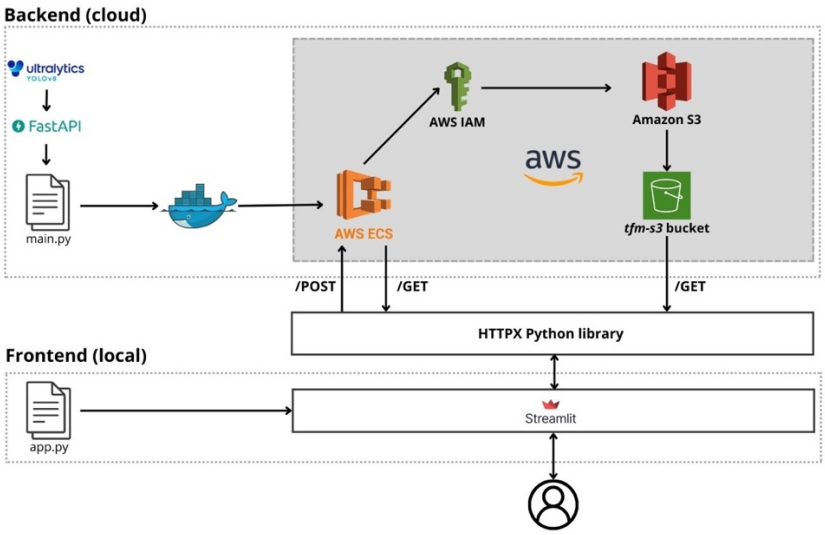
Containerization and Image Management

- **Docker Containers / Docker Hub**: To ensure environment consistency and simplify deployment, the Backend was containerized using Docker. The resulting image, which encapsulates all dependencies and the model weights, was pushed to my public repository on Docker Hub (/rogergs94). This allows the cloud infrastructure to pull and deploy the exact same version of the system during every execution cycle (click here to go to my repository: https://hub.docker.com/repository/docker/rogergs94/tfm_capstone_constructeye_ai/general)

Cloud Infrastructure (AWS Cloud)

To guarantee system scalability and availability, the architecture is deployed on Amazon Web Services (AWS) using a containerized approach:

- **AWS ECS (Elastic Container Service) with Fargate**: Both the Backend and Frontend are packaged as Docker images (stored in Docker Hub) and executed via AWS Fargate. This serverless modality allows for container management without the need to administer physical servers, scaling resources according to video processing demands.
- **AWS S3 (Simple Storage Service):** Utilized as the persistent object storage system. When the /process-video_df endpoint generates a result, the processed video and infringement captures are automatically stored in an S3 bucket. This ensures that industrial safety evidence is not lost upon container restarts and facilitates subsequent downloads via the /download-video endpoint.
- **Security and Networking**: Component interaction is secured through Security Groups and IAM (Identity and Access Management) roles, granting the FastAPI API exclusive permissions to write to the S3 storage.



You can clone my repository here:

https://github.com/rogergs94/tfm_capstone_constructeye_ai/tree/main/tfm_capstone_constructeye_ai

To run my application, just clone my repository and follow the steps in the annexes. Also, you can try the first version of my app by clicking here. You can also download the V2 **here**

8. Conclusions and Future Lines of Work

The development of this safety monitoring system for construction environments has validated the effectiveness of integrating Deep Learning models with modern and efficient deployment architectures. Upon completion of the project, the following conclusions can be drawn:

- Technical Effectiveness: The Spatial Exclusion Logic successfully addressed the data scarcity for the non-PPE classes, transforming a model limitation into a systemic strength through geometric reasoning.
- Operational Viability: The implementation of frame_stride demonstrated that high-complexity computer vision solutions can be executed on standard hardware without compromising safety standards.
- Business Value: The project transcends simple visual detection, evolving into a prevention and auditing tool capable of generating digital evidence for on-site risk management.
- While the YOLOv8 architecture belongs to an earlier generation, its performance remains competitive with more recent iterations like YOLO11, proving its reliability in production environments.

Future Lines of Work

Despite the positive results, the system offers several opportunities for expansion and technical improvement in future iterations.

- Future work should explore the integration of emerging models, such as potential YOLO25/26 releases, to benchmark improvements in inference speed and detection accuracy.

- Edge Computing: Deploying the system on devices such as NVIDIA Jetson or Raspberry Pi with AI accelerators, allowing inference to be performed directly on-site cameras without dependency on external servers.
- Automated Alerting: Integration with messaging channels for instantaneous notifications of critical safety violations.
- Predictive Analytics: Using the historical violation data to predict high-risk zones and timeframes through ML models.
- Class Expansion: The current model can be expanded to detect additional safety classes such as machinery.

9. Bibliography

- New York City Department of Buildings. (n.d.). Construction-related incidents [Data set]. Data.gov. Retrieved December 7, 2025, from https://catalog.data.gov/dataset/construction-related-incidents

- Trout, D., Earnest, G. S., Pan, C., & Wu, J. Z. (2022, 10 de noviembre). Head protection for construction workers: What we know. NIOSH Science Blog. https://blogs.cdc.gov/niosh-science-blog/2022/11/10/construction-helmets/

- Goodfellow, I., Bengio, Y., y Courville, A. (2016). The Deep Learning Book. MIT Press. http://www.deeplearningbook.org

- Ultralytics (2025). Everything You Need to Know About Computer Vision in 2025. Ultralytics Blog. https://www.ultralytics.com/blog/everything-you-need-to-know-about-computer-vision-in-2025

- V7 Labs. (2022, 21 de noviembre). YOLO Object Detection: A Complete Guide. V7 Blog. https://www.v7labs.com/blog/yolo-object-detection

- Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 779-788. https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Redmon_You_Only_Look_CVPR_2016_paper.pdf

- Ruman. “YOLO Data Augmentation”. Medium. https://rumn.medium.com/yolo-data-augmentation-explained-turbocharge-your-object-detection-model-94c33278303a

Annexes

Downloading Dataset

```
#I'm creating an environment variable for the API_key
API_key = userdata.get('ROBOFLOW_API_KEY')
rf = Roboflow(api_key = API_key)
```

```
project = rf.workspace("rogergs94").project("tfm_construction_roger")
version = project.version(3) #Version = 3
dataset = version.download("yolov8")
```

```
loading Roboflow workspace...
loading Roboflow project...
Downloading Dataset Version Zip in tfm_construction_roger-3 to yolov8:: 100%|| 337161/337161 [00:12<00:00, 27490.36it/s]

Extracting Dataset Version Zip to tfm_construction_roger-3 in yolov8:: 100%|| 6360/6360 [00:01<00:00, 4491.27it/s]
```

```
# Where the dataset has been downloaded?
base_path = dataset.location
print(f"Dataset path: {base_path}")
```

```
Dataset path: /content/tfm_construction_roger-3
```

EDA

Creating the df (training folder) for EDA

```
# --- CONFIGURATION ---
path_labels = f'{base_path}/train/labels'
class_names_dict = {i: name for i, name in enumerate(class_names)}

# --- DATA EXTRACTION ---
data_list = []

for label_file in glob.glob(os.path.join(path_labels, "*.txt")):
    # It will extract the file name (ie: "image_01") without the .txt extension
    file_name = os.path.splitext(os.path.basename(label_file))[0]

    with open(label_file, 'r') as f:
        lines = f.readlines()
        instances_in_image = len(lines)
        for line in lines:
            parts = line.split()
            if len(parts) == 5:
                cls_id = int(parts[0])
                name = class_names_dict.get(cls_id, f"ID_{cls_id}")

                data_list.append({
                    'image_name': file_name,
                    'class': name,
                    'x': float(parts[1]),
                    'y': float(parts[2]),
                    'w': float(parts[3]),
                    'h': float(parts[4]),
                    'area_pct': float(parts[3]) * float(parts[4]) * 100,
                    'instances_per_img': instances_in_image
                })

# Transformation into a dataframe
df = pd.DataFrame(data_list)
df.head()
```

	image_name	class	x	y	w	h	area_pct	instances_per_img
0	ppe_0985_jpg.rf.1b05922b87c99e6cb46645c3c8cd91a9	vest	0.815430	0.521484	0.363932	0.357910	13.025506	9
1	ppe_0985_jpg.rf.1b05922b87c99e6cb46645c3c8cd91a9	vest	0.487956	0.490723	0.367188	0.330078	12.120056	9
2	ppe_0985_jpg.rf.1b05922b87c99e6cb46645c3c8cd91a9	vest	0.200521	0.472412	0.332031	0.289551	9.613991	9
3	ppe_0985_jpg.rf.1b05922b87c99e6cb46645c3c8cd91a9	person	0.758464	0.582275	0.483073	0.835449	40.358289	9
4	ppe_0985_jpg.rf.1b05922b87c99e6cb46645c3c8cd91a9	person	0.468424	0.583984	0.560547	0.832031	46.639252	9

Image Analysis. Instances per image (training df)

```
# Instances per image in training
df_instances_per_img = df.groupby(by="image_name").agg(
    {"class":"nunique",
     "instances_per_img":"max"}
).sort_values(by="instances_per_img", ascending=False).reset_index()

df_instances_per_img.head()
```

	image_name	class	instances_per_img
0	ppe_0604_jpg.rf.dcd50aa024e76d261e4ae44536f154dd	4	39
1	ppe_0604_jpg.rf.fada3f91c5648270c83457b605c0b755	4	39
2	ppe_0604_jpg.rf.fe923e56ca7c0dd4236d5e2d7ed1edd6	4	39
3	ppe_0444_jpg.rf.4dcc3091adf3a2768f0f147a0db5f255	3	34
4	ppe_0444_jpg.rf.8093f0251bdb4111a26fd273ea254016	3	34

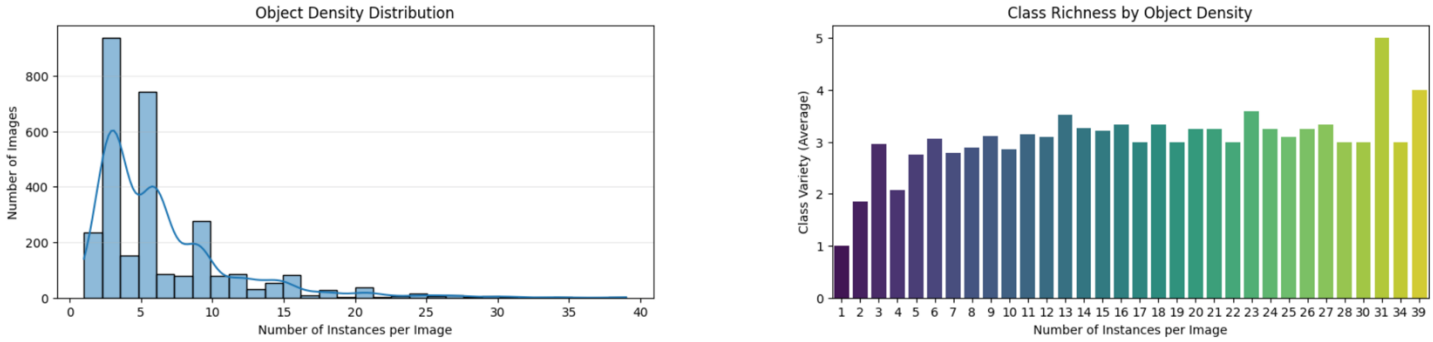
Image Analysis. Class diversity

```
# How many images there are per "instances_per_img" (total bboxes)?
df_density = df_instances_per_img.groupby(by="instances_per_img").agg(
    {"image_name":"count"}
).sort_values("image_name",ascending=False)

# Group by instances_per_img and calculate the mean class diversity for each group.

df_density["avg_diversity"] = df_instances_per_img.groupby("instances_per_img")["class"].mean()
df_density = df_density.rename(columns={"image_name": "total_images"})
df_density = df_density.reset_index()

df_density.head(10)
```

	instances_per_img	total_images	avg_diversity
0	3	936	2.958333
1	6	597	3.068677
2	9	276	3.115942
3	2	205	1.858537
4	4	153	2.078431
5	5	145	2.751724
6	12	85	3.105882
7	7	84	2.785714
8	8	80	2.887500
9	15	65	3.215385

```python
# --- VISUALIZATION ---
fig, axes = plt.subplots(1, 2, figsize=(20, 4))
plt.subplots_adjust(wspace=0.3)

# PLOT 1: Object Density (What you already had)
sns.histplot(df_instances_per_img['instances_per_img'], ax=axes[0], bins=30, kde=True, palette='skyblue')
axes[0].set_title('Object Density Distribution', fontsize=12)
axes[0].set_xlabel('Number of Instances per Image')
axes[0].set_ylabel('Number of Images')
axes[0].grid(axis='y', alpha=0.3)

# PLOT 2: Class Diversity (Strategy Justification)
sns.barplot(data=df_density, x='instances_per_img', y='avg_diversity', ax=axes[1], palette='viridis')
# Note: Fixed the title target for axes[1]
axes[1].set_title('Class Richness by Object Density', fontsize=12)
axes[1].set_xlabel('Number of Instances per Image')
axes[1].set_ylabel('Class Variety (Average)')

plt.show()
```

```
Ignoring `palette` because no `hue` variable has been assigned.

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```



Instance Analysis. Df with bbox size and area average

```python
df.groupby(by="class").agg(
    {"x": "mean",
    "y": "mean",
    "w": "mean",
    "h": "mean",
    "area_pct":"mean",
    "instances_per_img":"mean"}
).round(4)
```

class	x	y	w	h	area_pct	instances_per_img
helmet	0.4997	0.3200	0.1233	0.1329	2.3088	10.2315
no-helmet	0.4722	0.3495	0.0870	0.1048	1.1612	12.5036
no-vest	0.4981	0.5799	0.2122	0.3292	8.6798	9.8192
person	0.4997	0.5608	0.2427	0.6082	17.7958	9.8106
vest	0.5029	0.5581	0.1896	0.3340	7.9311	10.6073

Model Comparison

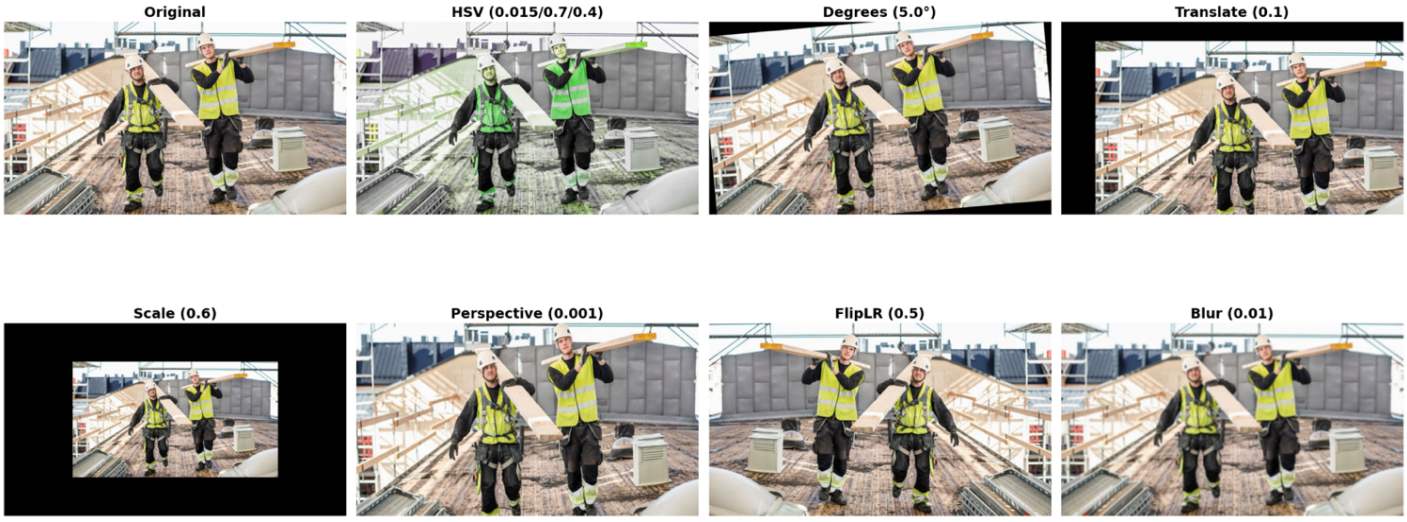
<u>Model Comparison (same code but changing the model name)</u>

```python
from ultralytics import YOLO

# Load a pretrained YOLO model (recommended for training)
base_path = dataset.location
model_name = "yolov8n"
model = YOLO(f"{model_name}.pt")

# 2. Name and project
project=f"TFM_{model_name}_{today}_no_Data_Aug"
name=f"modelo_{model_name}_50epochs"

# 3. Training
results = model.train(
    #data="/content/Hard-Hat-Workers-2/data.yaml",
    data=f"{base_path}/data.yaml",
    epochs=50,
    seed=42,
    patience = 5, #EarlyStopping
    imgsz=640,
    project=project,       # Carpeta principal
    name=name,   # Nombre de este entrenamiento
    save_period=1, #guarda los checkpoints en cada epoca
    #save_best=True
)

# 4. Validation
results = model.val()

# 4. Evaluation Metrics
print(f"mAP@50: {results.box.map50}")
print(f"mAP50-95: {results.box.map}")
print(f"Recall global: {results.results_dict['metrics/recall(B)']}")

# Recall per class:
for i, name in enumerate(results.names.values()):
    print(f"Recall de la clase {name}: {results.class_result(i)[2]}")
```

Oversampling

```
import os
import shutil
from tqdm import tqdm

# Original paths
train_labels_path = f'{dataset.location}/train/labels/'
train_images_path = f'{dataset.location}/train/images/'

# #End path
# It redirects to the train_oversampled directory and creates the images and labels subdirectories
oversampled_path = f'{dataset.location}/train_oversampled'
os.makedirs(f'{oversampled_path}/images', exist_ok=True)
os.makedirs(f'{oversampled_path}/labels', exist_ok=True)

# It takes the IDs based on the data.yaml file
# 1: no-helmet, 2: no-vest
clases_criticas = ['1', '2']

print("Initing balancing process...")

for label_file in tqdm(os.listdir(train_labels_path)):

    # 1. Determine image filename (preserving original extension)
    # Search for an image file that matches the label filename
    base_name = os.path.splitext(label_file)[0]
    img_file = None
    for ext in ['.jpg', '.jpeg', '.png', '.JPG']:
        if os.path.exists(os.path.join(train_images_path, base_name + ext)):
            img_file = base_name + ext
            break

    if img_file is None: continue # Skip if no matching image is found

    # 2. Read labels to determine the duplication factor
    with open(os.path.join(train_labels_path, label_file), 'r') as f:
        lines = f.readlines()
        has_violation = any(line.split()[0] in clases_criticas for line in lines)

    # 3. Define factor: 3 copies for critical images, 1 copy for normal images
    factor = 3 if has_violation else 1

    for i in range(factor):
        new_name = f"aug_{i}_{base_name}"

        # Copy image file
        shutil.copy(os.path.join(train_images_path, img_file),
                    os.path.join(oversampled_path, f"images/{new_name}{os.path.splitext(img_file)[1]}"))

        # Copy label file
        shutil.copy(os.path.join(train_labels_path, label_file),
                    os.path.join(oversampled_path, f"labels/{new_name}.txt"))

print(f"\nOversampling completed. New dataset located at: {oversampled_path}")
```

Data Augmentation (Albumentation Python library):



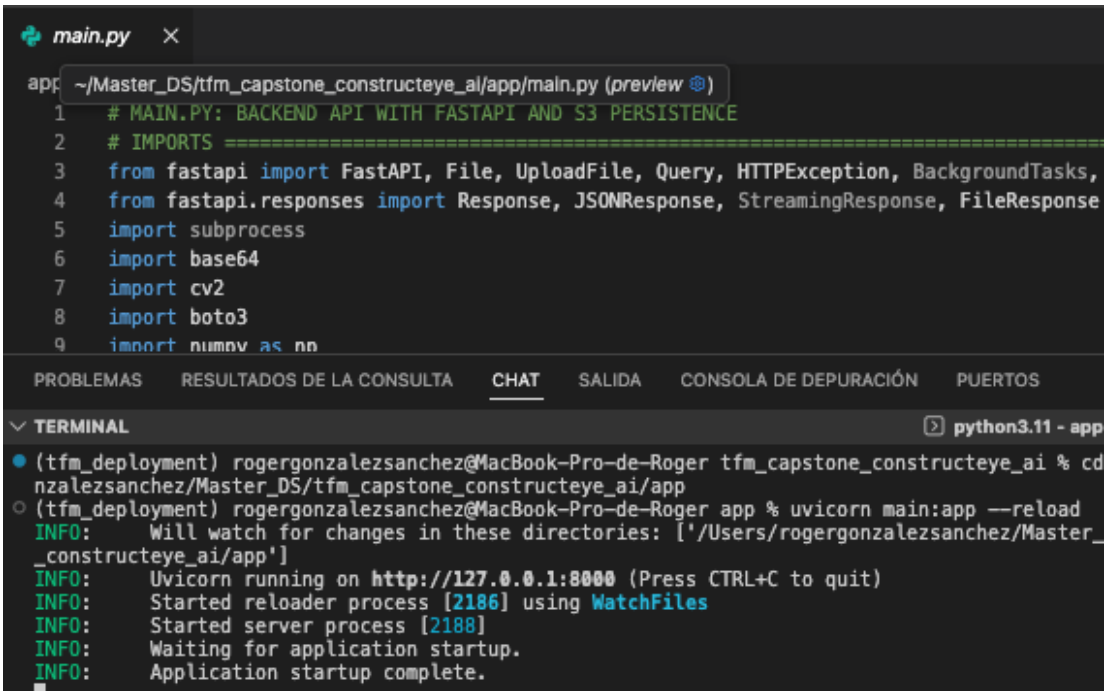
Training Model PostProcessing

```
# Training with Data Augmentation (YOLO PIPELINE) - YOLOv8n
model_name = "yolov8n"
model = YOLO(f"{model_name}.pt")

# Project naming for tracking experiments:
project=f"TFM_{model_name}_{today}_WITH_Data_Aug_1280"
name=f"modelo_{model_name}_optimized_50epochs_1280"

# Model training
model.train(
    data=f'{dataset.location}/data.yaml', # Path to the configuration file defining class names and dataset paths.
    project=project,                      # Name of the main project directory to group related experiments.
    name=name,                            # Specific identifier for this run (creates a subfolder within the project).
    epochs=50,                            # Maximum number of complete passes through the training dataset.
    patience=15,                          # Early Stopping: halts training if no improvement is seen after 15 epochs.
    seed=42,                              # Sets a fixed random seed to ensure experimental reproducibility.
    imgsz=640,                            # Input image resolution; balances detection detail with computational cost.
    freeze=0,                             # Number of initial layers to freeze; 0 indicates full-network fine-tuning.
    box=5,                                # Weight gain for the bounding box regression loss in the total cost function.
    save_period=1,                        # Frequency (in epochs) at which model checkpoints are saved.
    lr0=0.001,                            # Initial learning rate for the optimizer's weight update steps.
    optimizer="AdamW",                    # Optimization algorithm with decoupled weight decay for better generalization.
    label_smoothing=0.1,                  # Regularization that prevents overconfidence by softening target labels.
    weight_decay=0.0005,                  # L2 penalty applied to weights to prevent growth and reduce overfitting.
    # Data augmentation:
    **augmentation_params                 # Unpacks and applies stochastic transforms (Mosaic, Mixup, etc.) to the pipeline.
)

# Model validation
results_data_augm = model.val()

# Performance Metrics
print("\n--- FINAL METRICS (Optimized Model) ---")
print(f"mAP@50: {results_data_augm.box.map50:.4f}")
print(f"mAP50-95: {results_data_augm.box.map:.4f}")
print(f"Recall global: {results_data_augm.box.mr:.4f}") # mp=mean precision, mr=mean recall

print("\n--- CLASS-SPECIFIC RECALL ---")
for i, class_name in enumerate(results_data_augm.names.values()):
    # Index [1] of class_result corresponds to Recall
    recall_class = results_data_augm.box.class_result(i)[1]
    print(f"Class Recall {class_name}: {recall_class:.4f}")
```

Deployment

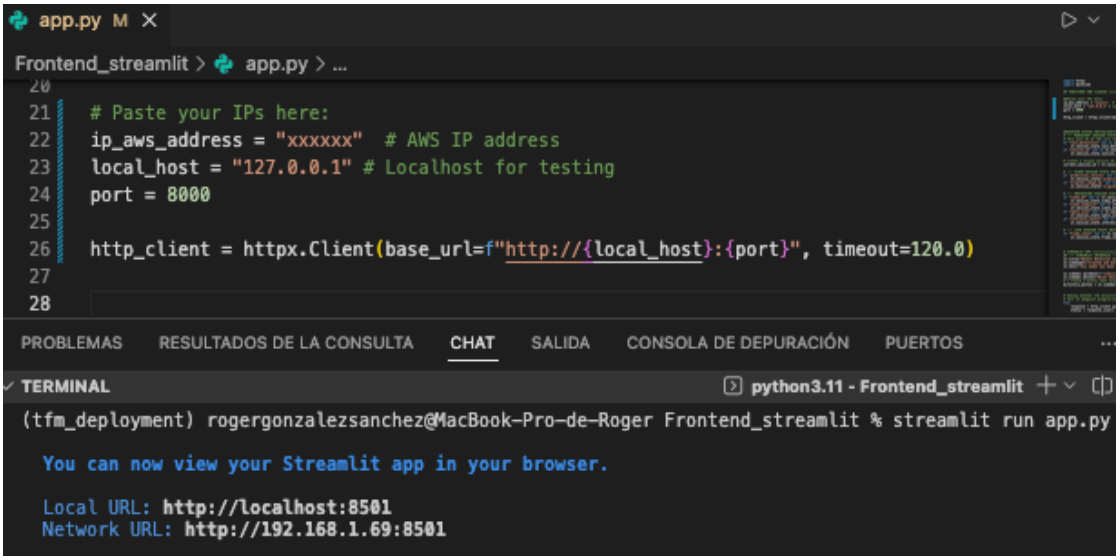
1. ***Running main.py in with Uvicorn (local)***

   Go to the “app” directory in Visual Studio Code and run in the terminal: **uvicorn main:app --reload**



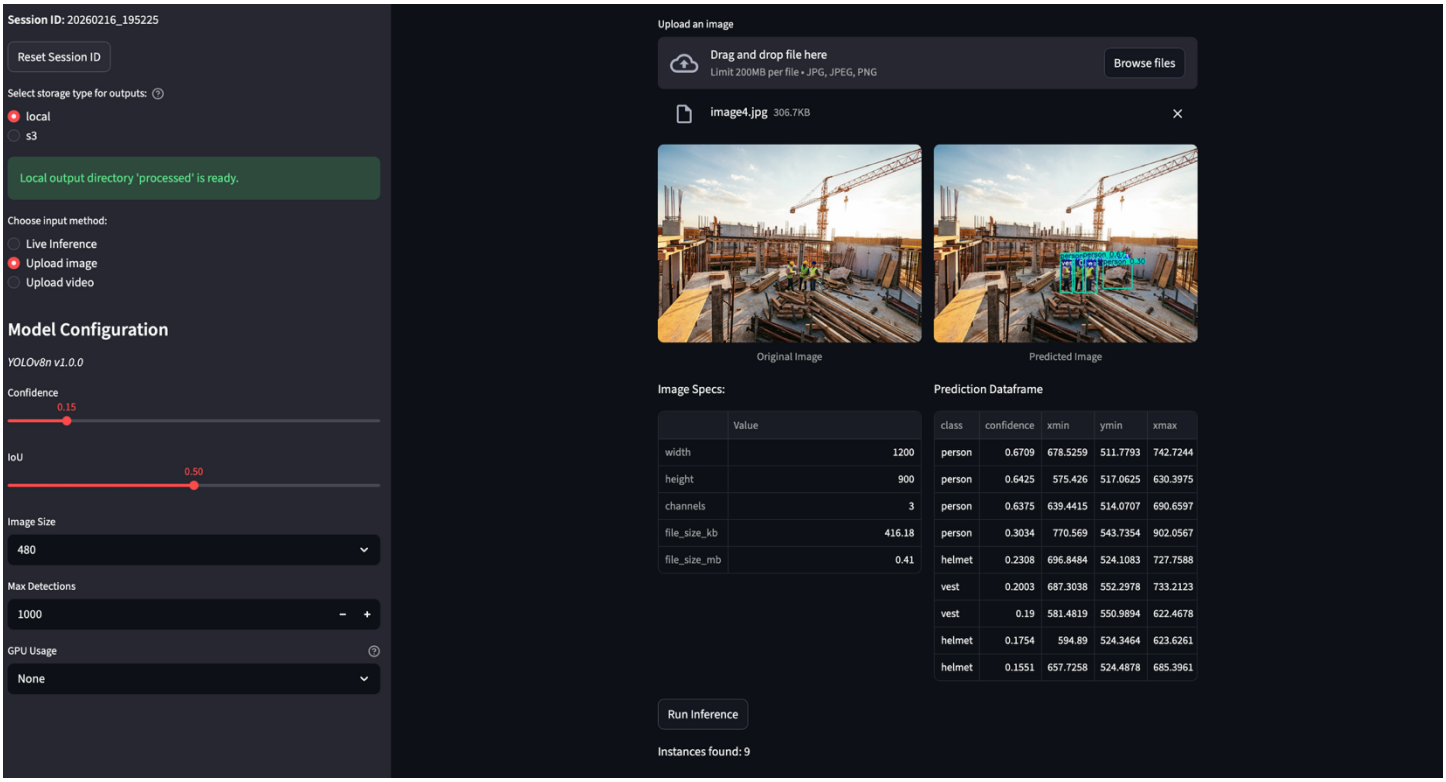
2. ***Running app.py (frontend) with Streamlit in local***

   Go to the “Frontend streamlit” directory and run a new terminal in VSC. Then, go to the directory using “cd” and input **streamlit run app.py.** Please note that you must have the Streamlit library installed

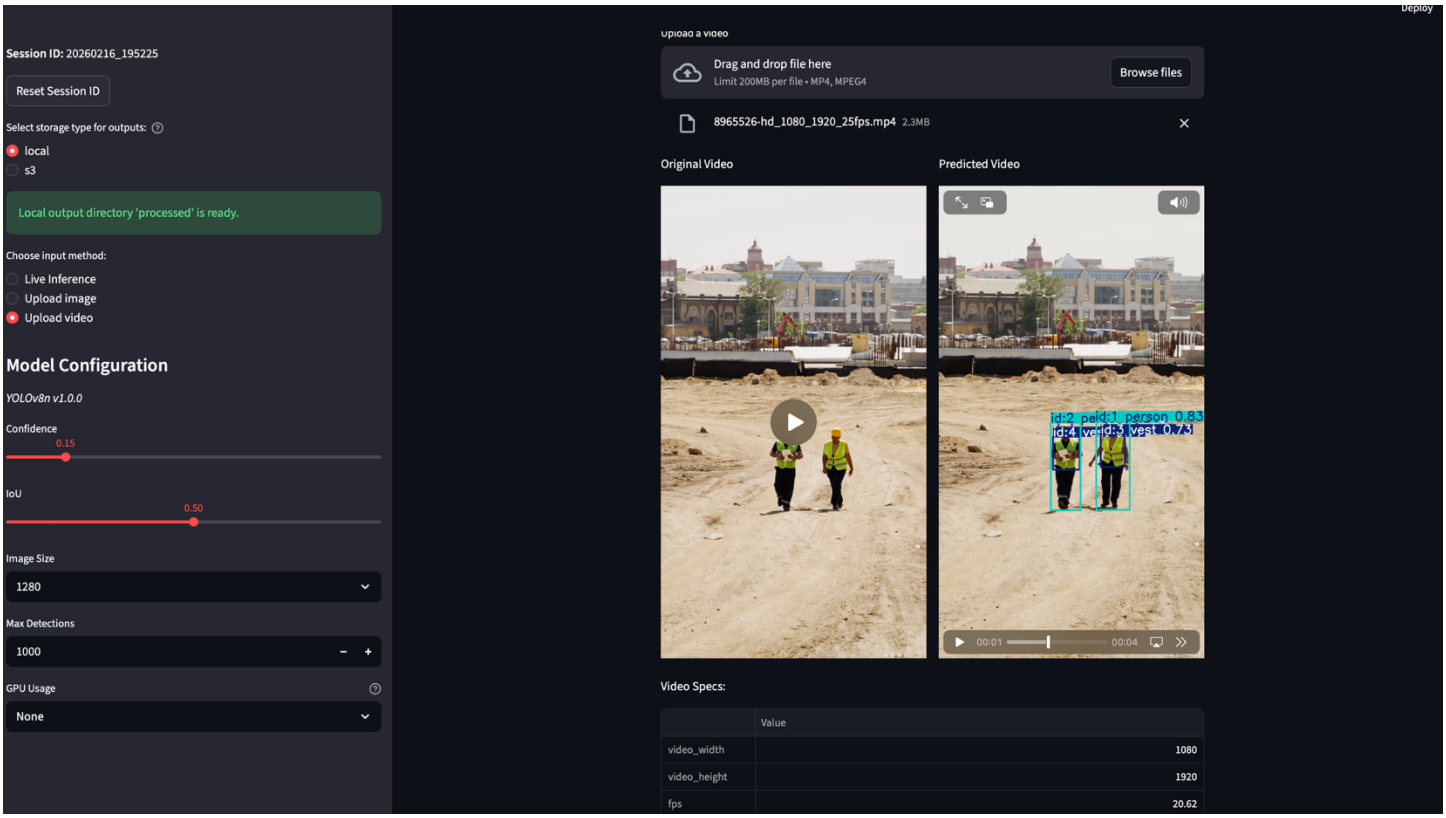


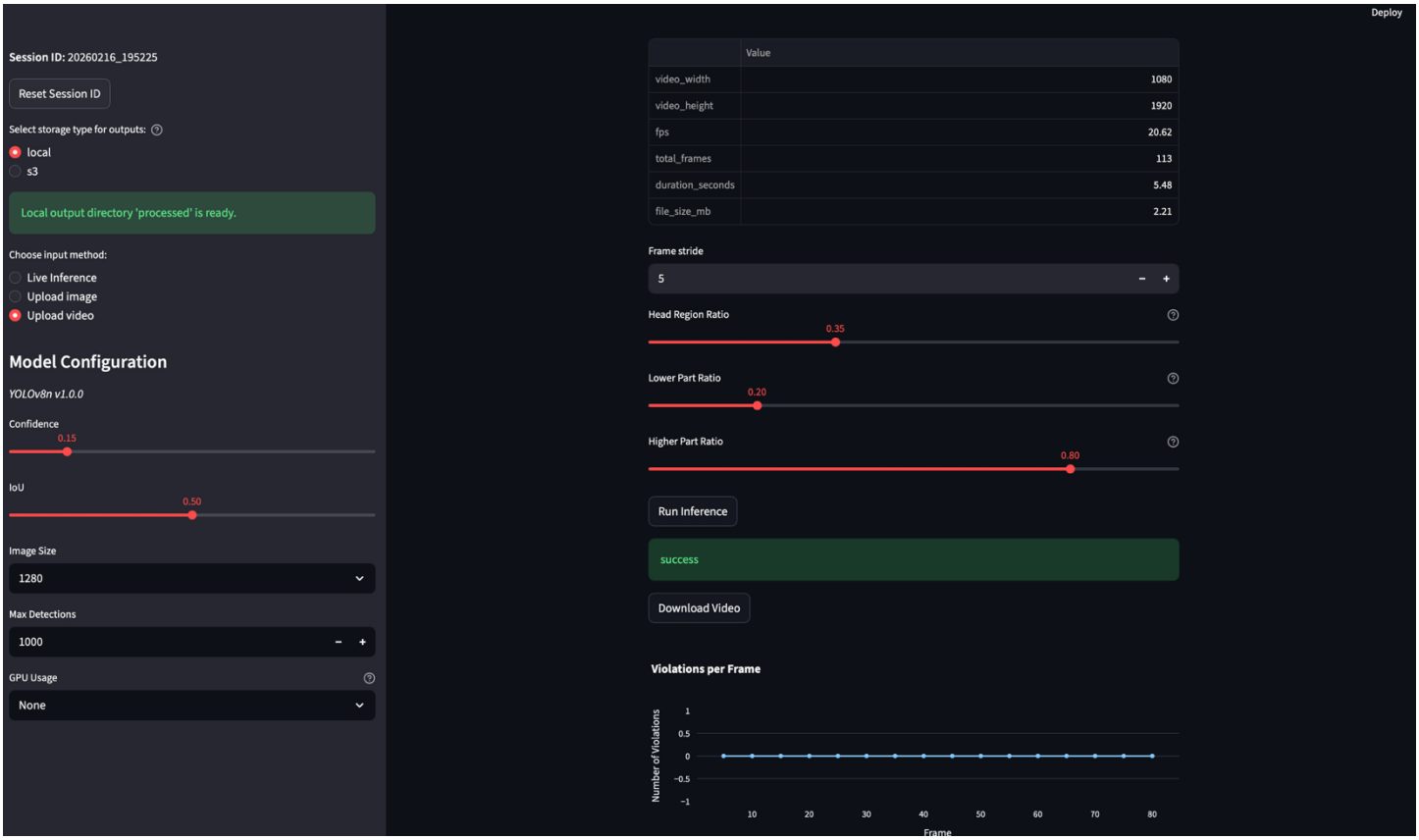
Running app.py (frontend) with Streamlit

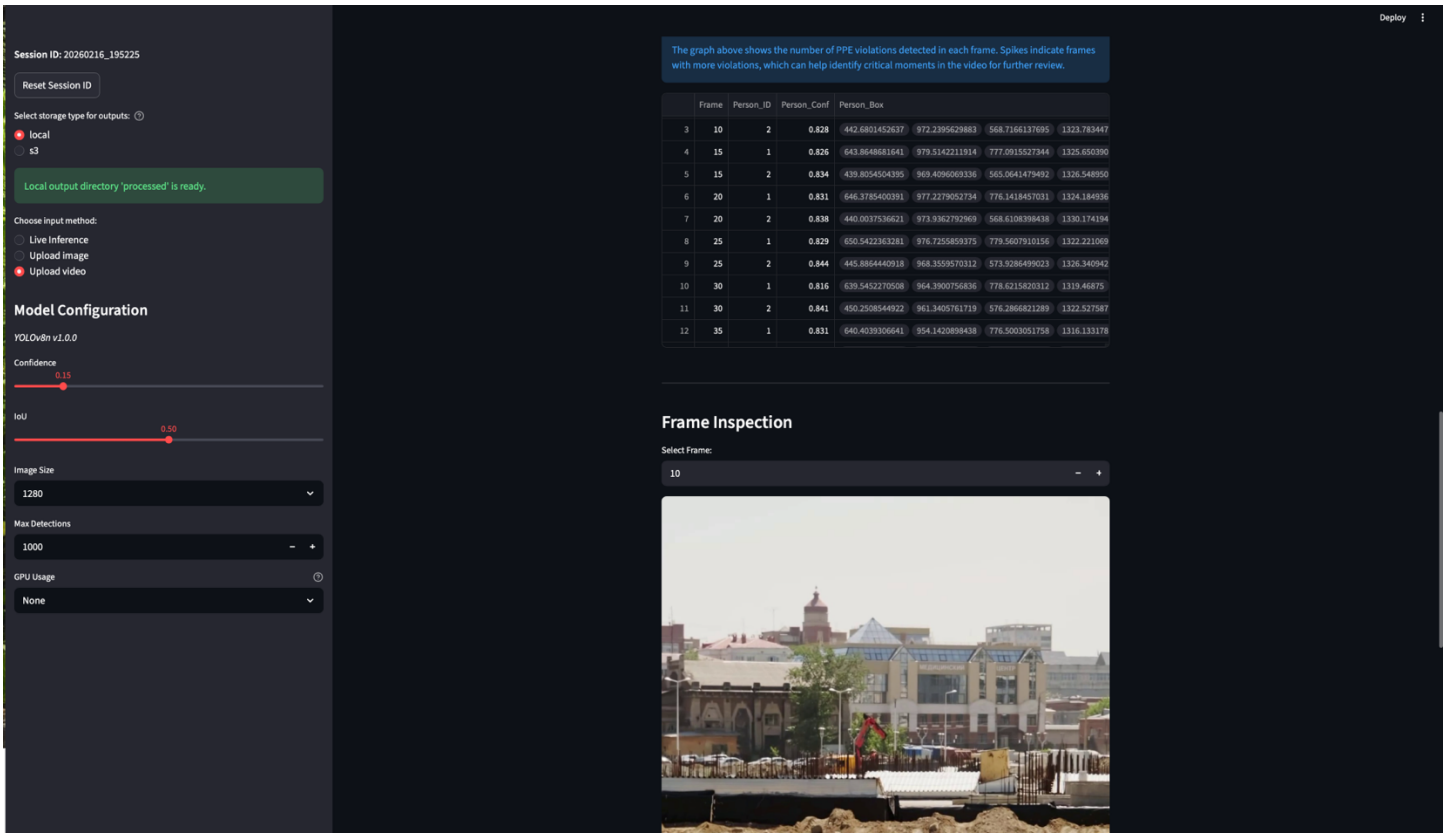
<u>Image inference</u>



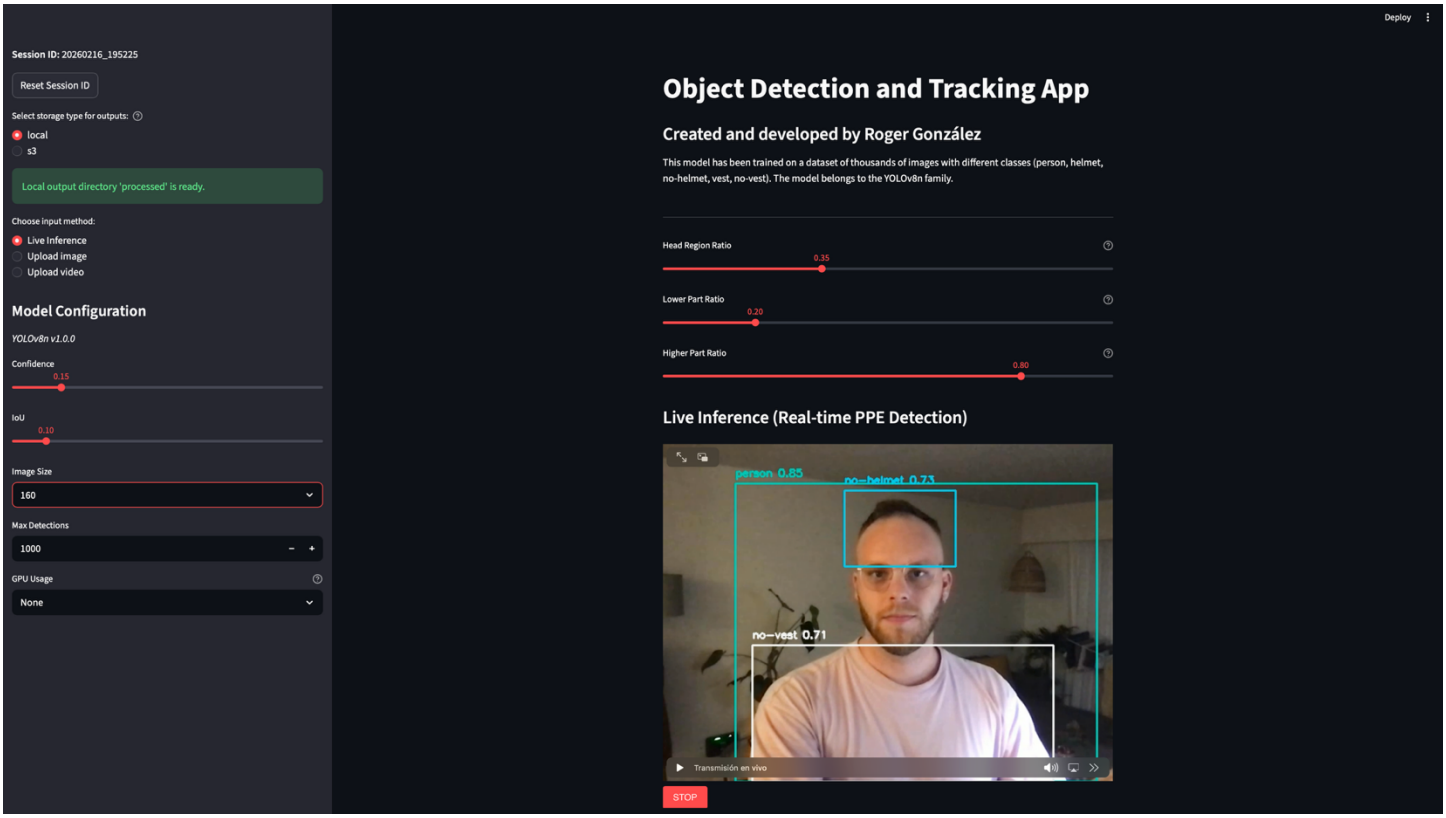
Video inference







Live Inference



Extra. Genetic Hyperparameter Tuning (Grid Search)

```
# Define search space
search_space = {
    # Optimization hiperparamst
    "lr0": (1e-5, 1e-1),
    "momentum": (0.7, 0.937),
    "weight_decay": (0.0, 0.001), # Regularization to avoid overfitting

    # Data Augmentation hiperparams
    #"copy_paste": (0.6, 0.8),
    #"mixup": (0.15, 0.3),
    #"degrees": (0.0, 45.0),
    #"cls": (1.0, 2.5),
}
```

```
# Initialize the YOLO model

path_best_pt = f"/content/runs/detect/{project}/{name}/weights/best.pt"
path_best_pt = "/content/best.pt"
model = YOLO(path_best_pt)


# Grid search
model.tune(
    data=f"{dataset.location}/data.yaml",
    epochs=20,
    seed=42,
    iterations=100,
    optimizer="AdamW",
    space=search_space,
    plots=False,
    save=False,
    val=False,
)
```