Driblab Take-Home Task Report

Luis Carlos Catalá Martínez

As part of the selection process for the position of Junior Machine Learning position at Driblab, I was given a take-home task. In this task I was asked to, for each fifth frame of 3 football match clips, locate and count the players of each of the two teams and referees, as well as locating the ball.
This report summarizes the development process of a piece of software that achieves such a goal, explaining the steps taken and key decisions made along the way.

Technologies used

- As requested, all the code was written in Python
- For reading, writing and manipulating images and videos, I used OpenCV and NumPy
- For modeling I used both Scikit-Learn and PyTorch, as well as Ultralytics as a way of accessing Yolo v8
- A slew of additional and typical python libraries, such as tqdm and loguru, for quality of life

Input

I used OpenCV's VideoReader to get the selected clip's frames into a pre-allocated NumPy array, thus preventing the need to append them and its additional computational costs.

Player and Referee detection

Person detection

The first step in this process is to get the locations of people in the playing field, a typical object detection task. For this we'll use the YOLO v8 model, since we're allowed any YOLO version. This model, available through the *ultralytics* python package, provides several architectures with varying amounts of weights. Each one strikes a different balance between performance and accuracy (mean Average Precision in this case). Based on this table from their Github repository, I opted for their "m" variant, since it offers the most balance between the two. Smaller models had bounding boxes varying too much in size, and larger models were too slow for the mAP gains offered.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

The model can be configured to only return detections of class "person" (and "sports ball" as we'll see later).
Since the model only shows detections of people on the field, not giving us false positives from the stadium steps, we don't need to filter them out. A good way of doing so though would be to check if the bounding box also contains a good amount of green pixels, since that means they have grass as their background.

Referee discrimination

Since the object detection model only tells us if an object is a person or a ball, we can't tell if they're from either of both teams or a referee, I added a few extra steps to discern them., starting with checking if a person is a referee. For this I checked the amount of "nearly black" pixels in a person's bounding box.
To do this I converted the image from BGR to greyscale, and checked the percentage of pixels with a value above 30. If this was above 7%, the player is considered a referee. I found these values by playing around with the parameters until finding a result I liked, with little false positives and negatives.
The biggest drawback from this method are situations like we find in clip 3, where we get many false positives when players move into shade. To remedy this, I believe a complete rework of the method is needed.
Finally, we draw a black bounding box with the coordinates given by the detection model for this person.
Converting the image to HSV and filtering based on the V value didn't seem to offer any advantages, and neither did applying a histogram normalization to the grayscale image beforehand.

Player team discrimination

The next step is discerning the team each player belongs to. Since their most distinguishing feature is the most prevalent color of each player, I ended up using solely this.
I extracted each bounding box' most prevalent color by:

1. Removing any green pixels in the HSV space,
   specifically in the range [(70, 0, 50), (80, 255, 120)], and turning them to black.
2. Calculating its histogram and getting the bin with the highest count, ignoring black (0,0,0).
3. Convert colors to BGR
4. In the first frame, use all the BGR colors to train a KMeans model with 2 clusters.
5. On all frames, classify all the player bounding boxes with the KMeans model.
6. Use the cluster centers as team colors to draw player bounding boxes in the output video
7. Count the players of each class and write them to the output JSON. There is no detection of which team

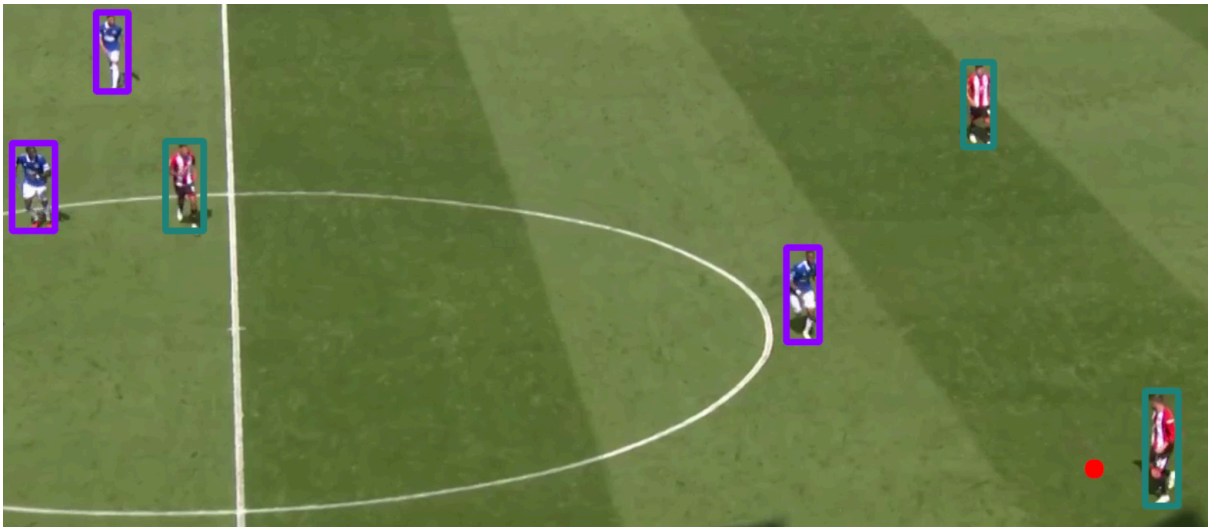
I also tried to use the HSV colors directly for the KMeans model, but couldn't get it to work better than with the BGR format.
Using the average of each team's prevalent colors offers a more stable color for each team than tinting each bounding box with its player's prevalent color, making it more robust to lighting or pose changes.
My first approach for this step involved training a convolutional network with an encoder-decoder architecture, and using the encoder's features to then classify the players in 3 classes (2 teams and referees). Trying both my own network and a pretrained ResNet50, in both cases the result was too slow (taking about a second per frame in my local machine) and discarded it before getting it to work.

Ball detection and size estimation

The next step is to locate the ball in each frame and estimate its size. In this part I followed these steps:

1. Add the ball class to the inference parameters of the same object detector used for the persons.
2. Since there can only be a single ball at a time, select the bounding box with the highest confidence in case it detected more than 1.
3. From the selected bounding box, I took the center X and Y coordinates as the ball's position and the box' width and height as the ball's width and height. If in a given frame the ball is not detected, its last known position and size is used. If it's not detected at the first frame, we will return a 0 for all of them.

My original idea was to get a better ball size estimation by applying either a color mask to the image inside the bounding box, and getting the actual center and sizes from that. However, the bounding box the model returns fits the ball so well that there's no need to do any additional processing.
Finally, the ball's position is annotated as a red dot on the image when it is detected, and the ball's X, Y, width and height are written to the output JSON contents.
The biggest drawback of this method is a lack of detections of the ball. For this I recommend using one of the larger YOLOv8 architectures and take the performance penalty (pun not intended).



Output

This program's output is twofold:

1. A .mp4 video file with all the annotations of players and referees bounding boxes and the ball's position. This is written with OpenCV's VideoWriter.
2. A .json file with annotations of each team's players and referee counts and the ball's position and size every 5 frames. Its contents are generated using f-strings and the string.join() method, since they are considered the fastest string manipulation methods in python, and finally written as a regular text file.

Future lines of work

Due to a lack of time, there are some areas that could be improved upon or where I could have dove deeper, these being:

- Better referee recognition: the current method is not robust against changes in lighting. I imagine the best way of doing this is comparing each bounding box' similarity to each of the cluster centers of the KMeans as well as checking the percentage of dark pixels, giving us a better understanding of whether the person is likely to be a player apart from just a referee.

- Better ball detections: the model architecture I selected for detecting players isn't enough for detecting the ball all the time, especially when it's moving behind a player's legs or it's shot really high up and has the spectators as background instead of grass. For this, I started developing a really simple predictive model but ended up not delivering it as it was requested to only annotate the ball when detected. For the prediction, even a simple estimation from the ball's previous known positions would improve the results, especially when the ball is rolling on the field. This involves taking the positions from the frames where the ball was detected and estimating a velocity per frame, then applying that same vector to the last known position. From what I saw in my testing it doesn't really work when the camera pans a lot, or there are multiple consecutive frames without ball detections.
- Improve team classification model: other clustering models aside from KMeans should be tested. Also it should be tested with different input data, such as the H channel in HSV color space. Also, the prevalent color detection algorithm could probably benefit from downsampling each player's cropped image.
- Further performance optimizations: there are probably redundant color space changes during the processing of each frame. These could probably be reduced by reorganizing my algorithm's structure. Since this is not a performance bottleneck (most of each frame's time is taken by the detection model), it wasn't worth it doing it at this stage.

Running the code

The project is provided in a zip file containing the python source code and a requirements.txt file with the necessary dependencies. To run it:

1. Uncompress the zip file
2. Create a Python virtual environment in the directory .venv with **python3 -m venv .venv**
3. Source the environment with **source .venv/bin/activate**
4. Run the script for video processing like
   python3 process_video.py <path_to_clip>
   for example, for clip 1, which is stored in “videos” folder:
   python3 process_video.py videos/clip1.mp4
   This will generate the requested video and json outputs with the names output.mp4 and output.json, but you can change them with the optional parameters **–json_output_path** and **–video_output_path**.