Forest Fire Risk Detection Using Computer Vision and Machine Learning

Jiesi Zhang, Jiannan Ye, Jianming Lu
Khoury College of Computer Sciences, Northeastern University, Vancouver, Canada
email: zhang.jies@northeastern.edu, ye.jian@northeastern.edu, lu.jianm@northeastern.edu

***Abstract*—Wildfires are among forces of nature that cause huge devastation to humans and the environment. The 2021 wildfire season in B.C. was the third-worst on record in terms of area burned. In this paper, we will show how to use YOLOv5 model and map coloring algorithm to estimate risk levels of the forest by recognizing alive trees and dead trees and help the B.C. government prevent the wildfires. After adjusting our models, the best mean average precision is 53.3 percent.**

***Keywords*—YOLO, Forest Fire, Object Detection, Computer Vision, Machine Learning**

I. Introduction

Wildfires are uncontrolled, rapidly spreading, and raging huge flames enhanced with wind action and firebrands that can wipe out an extensive forest or vegetation land area within minutes. Wildfires are among forces of nature that cause huge devastation to humans and the environment. With more than 1,600 fires burning nearly 8,700 square kilometers of land, the 2021 wildfire season in B.C. was the third-worst on record in terms of area burned. In total, the B.C. government has spent 565 million on fighting wildfires this year, and 181 evacuation orders were issued.

To help the B.C. government control and prevent wildfire in the future, we will try to use computer vision tools and machine learning to estimate risk level in given forest images. In the algorithm description section, we will discuss how the YOLOv5 model works. In the Methodology section, we will show how to create the YOLOv5 model and implement the map coloring algorithm step by step. Finally, in the result and conclusion section, we will talk about the final result and our thought about future research.

II. Algorithm Description

A. YOLO

YOLO (You Only Look Once) reframes object detection as a single regression problem rather than a classification problem. A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes shown in figure 1 [1]. YOLO reasons globally about the image when making predictions. It sees the entire image during training and testing time, so it implicitly encodes contextual information about classes and their appearance[1].

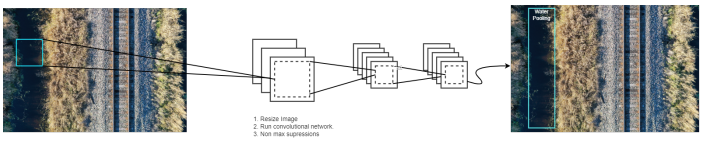

Fig. 1. YOLO Object Detection

1) Detection: The input image is divided into an $S * S$ grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting the Object. Each grid cell predicts B bounding boxes and confidence scores for those boxes. Confidence is defined as $P_r(Object) * IOU(truth, pred)$. If no object exists in that cell, the confidence score is zero. Otherwise, the confidence score equals the intersection over union (IOU) between the predicted box and the ground truth[1].

Each bounding box consists of 5 predictions: *x, y, w,h*, and *confidence*. The *(x,y)* coordinates represent the center of the box relative to the bounds of the grid cell. The *(w,h)* coordinates represent the width and height of the bounding box and are predicted relative to the whole image. Finally, the confidence prediction represents the intersection over union (IOU) between the predicted box and any ground truth box[1].

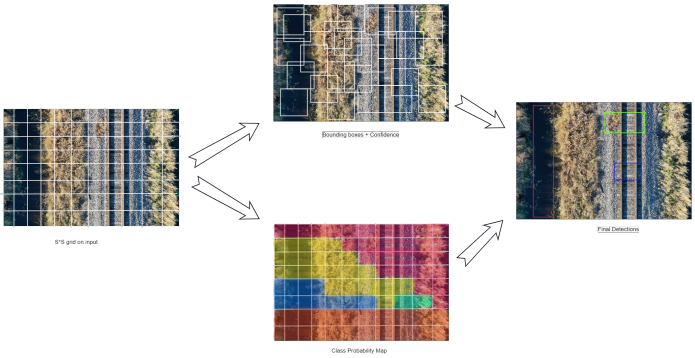

Fig. 2. Object Detection using YOLO

Each grid cell also predicts C conditional class probabilities, $P_r(Class_i \mid Object)$. These probabilities are conditioned on the grid cell containing an object. Only one set of class probabilities are predicted per grid cell regardless of the number of boxes. At test, the conditional class probabilities and the individual box confidences are multiplied as[1],

$$P_r\left(Class_i|Object\right) * P_r\left(Object\right) * IOU_{pred}^{truth} = P_r\left(Class_i\right) * IOU_{pred}^{truth} \quad (1)$$

2) Architecture: The model is implemented as a convolutional neural network. The initial convolutional layers of the network extract feature from the image, while the fully connected layers predict the output probabilities and coordinates. The detection network architecture has 24 convolution layers followed by two fully connected convolution layers.

978-1-7281-9656-5/20/$31.00 ©2020 IEEE

Alternating 1 * 1 convolution layers reduce the features space from preceding layers. The network is shown in figure 3.

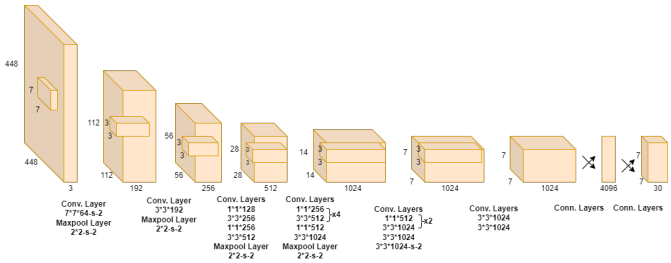

Fig. 3. Architecure of YOLO

3) Activation and Loss Function: The final layer predicts both class probabilities and bounding box coordinates. The bounding box coordinates are normalized to 0 and 1. A linear activation function for the final layer and all other layers use the following leaky rectified linear activation:

$$\phi(x) = \begin{cases} x, & if x > 0 \\ 0.1x, & otherwise \end{cases} \tag{2}$$

The sum-squared error in the output of the loss function is optimized using the equation below.

$$\begin{aligned} &\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left[(x_i - \widehat{x}_i)^2 + (y_i - \widehat{y}_i)^2 \right] + \\ &\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\widehat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\widehat{h}_i} \right)^2 \right] \\ &+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left(C_i - \widehat{C}_i \right)^2 + \\ &\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{noobj} \left(C_i - \widehat{C}_i \right)^2 + \\ &\sum_{i=0}^{S^2} 1_i^{obj} \sum_{c \varepsilon classes} (p_i(c) - \hat{p}_i(c))^2 \end{aligned} \tag{3}$$

B. YOLOv5

The architecture of YOLOv5 consists of three parts. Backbone: CSPDarknet, Neck: PANet, and Head: Yolo Layer.

The data are first input to CSPDarknet for feature extraction, and then fed to PANet for feature fusion. Finally, Yolo Layer outputs detection results (class, score, location, size) [2].

Joseph Redmon (inventor of YOLO) introduced the anchor box structure in YOLOv2 and a procedure for selecting anchor boxes of size and shape that closely resemble the ground truth bounding boxes in the training set. By using the k-mean clustering algorithm with different k values, the authors picked the 5 best-fit anchor boxes for the COCO dataset (containing

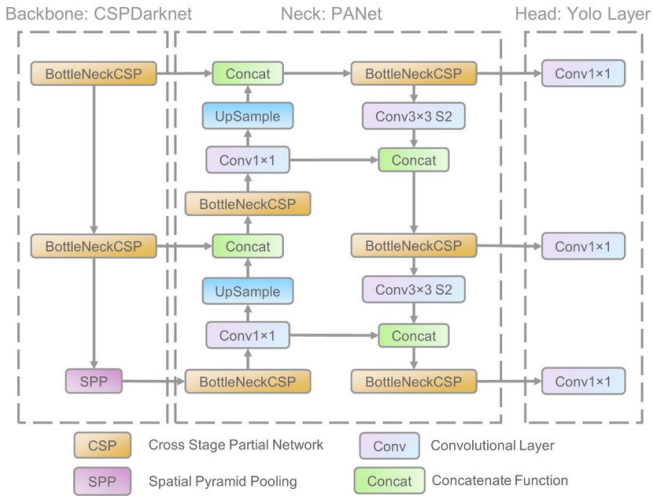

Fig. 4. Architecure of YOLOv5

80 classes) and use them as the default. That reduces training time and increases the accuracy of the network[3].

However, when applying these 5 anchor boxes to a unique dataset (containing a class not belonged to 80 classes in the COCO dataset), these anchor boxes cannot quickly adapt to the ground truth bounding boxes of this unique dataset. For example, a giraffe dataset prefers the anchor boxes with the shape thin and higher than a square box. To address this problem, computer vision engineers usually run the k-mean clustering algorithm on the unique dataset to get the best-fit anchor boxes for the data first. Then, these parameters will be configurated manually in the YOLO architecture[3].

Glenn Jocher (inventor of YOLOv5) proposed integrating the anchor box selection process into YOLOv5. As a result, the network has not to consider any of the datasets to be used as input, it will automatically "learning" the best anchor boxes for that dataset and use them during training[3].

III. Methodology

A. Deal with the original dataset

Building a good dataset is one of the most important ting for making a successful object detection project. We used Roboflow to annotate our original dataset, in a word, we need to make many labels for each picture.

Roboflow is a computer vision platform that enables users to build computer vision models faster and more accurately by providing better data collection, preprocessing, and model training techniques. Roboflow allows users to upload custom datasets, draw annotations, modify image orientation, resize images, modify image contrast, and perform data enhancements. It can also be used to train models.

Roboflow also has a universal annotation conversion tool that allows users to upload annotations and convert them from one format to another without having to script the conversion for a custom object detection dataset.

We totally annotated 589 pictures, which included 412 pictures for training set, 118 pictures for validation set and 59 pictures for testing set.

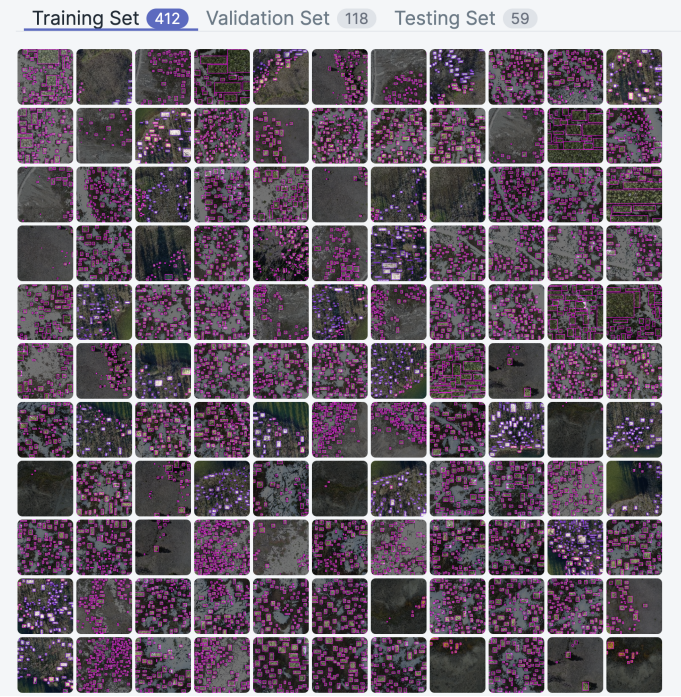

Fig. 5. Abstract of our annotation

Although it may be not regarded as a large dataset, but each picture had large number of annotations and we tried our best to annotate every tree.

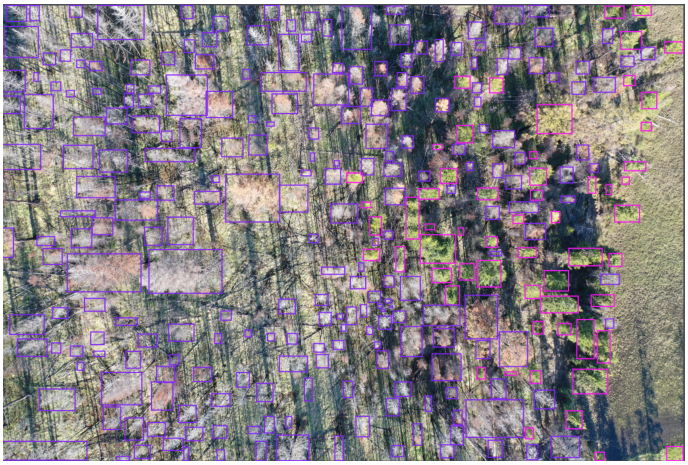

Fig. 6. Annotation Example

B. Model

We will show how to use the YOLOv5 model to implement alive tree and dead tree detection. We will walk through our code under the Google Colab environment. To help people who are not familiar with Python or YOLOv5 Model, I will show the code below to help them to reproduce my work.

1) Import packages and prepare the dataset: Firstly, we clone the YOLO project and install Roboflow as well as other dependencies.

```
#clone YOLOv5 and
!git clone https://github.com/ultralytics/yolov5  # clone repo
%cd yolov5
%pip install -qr requirements.txt # install dependencies
%pip install -q roboflow

import torch
import os
from IPython.display import Image, clear_output  # to display images

print(f"Setup complete. Using torch {torch.__version__} ({torch.cuda.get_device_properties(0).name if torch.cuda.is_available() else 'CPU'})")
```

Fig. 7. Import packages

Secondly, we create a Roboflow object and set the model as YOLOv5.

```
from roboflow import Roboflow
rf = Roboflow(model_format="yolov5", notebook="ultralytics")
```

Fig. 8. Create a Roboflow object

Next, we create the dataset directory environment.

```
# set up environment
os.environ["DATASET_DIRECTORY"] = "/content/datasets"
```

Fig. 9. Dataset directory environment

Then, we follow the Roboflow instruction to download our dataset by providing the API key.

```
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="VjyizEwlgGPXF07JMvyw")
project = rf.workspace("forest-fire").project("second-annotation-for-trees")
dataset = project.version(18).download("yolov5")
```

Fig. 10. Download the dataset

2) Train the model: Images are resized to 640 * 640 pixels because the model is suitable for training with an image size of 640*640 pixels. No augmentation will be used in this training. Since we have enough memory, the batch number we used is 32. To get the best precision, We used 500 as the epoch number. I used the YOLOv5m model because the dataset is relatively small. Other parameters are default in the YOLO model.

```
!python train.py --img 640 --batch 32 --patience 150 --epochs 500 --data /content/datasets/Second-Annotation-For-Trees-18/data.yaml --weights yolov5m6.pt --cache
```

Fig. 11. Train the model

Reading the model summary and assessing the performance

```
Model summary: 378 layers, 35254692 parameters, 0 gradients, 49.0 GFLOPs
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95:
                 all        118       8632      0.574      0.489      0.499      0.169
         Alive Tree         118       6995      0.628      0.612      0.625      0.235
          Dead Tree         118       1637       0.52      0.365      0.372      0.104
```

Fig. 12. Model summary

The result shows that the mean average precision (mAP@0.5) of all classes is 0.499. the mAP for the alive tree class and the dead tree class are 0.625 and 0.372.

3) Display the precision of images in the test dataset: the inference on testing images are shown as below:

```
#display inference on ALL test images

import glob
from IPython.display import Image, display

for imageName in glob.glob('/content/yolov5/runs/detect/exp2/*.jpg'): #assuming JPG
    display(Image(filename=imageName))
    print("\n")
```

Fig. 13. Check test images

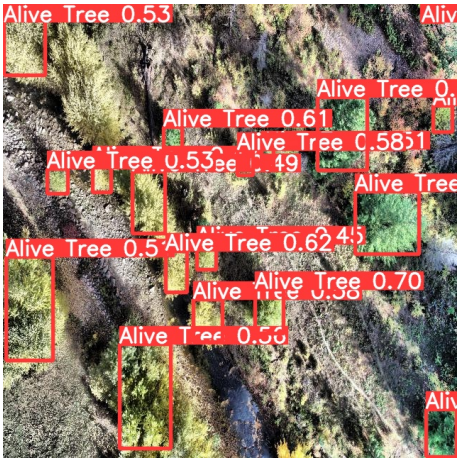

Fig. 14. Test image 1

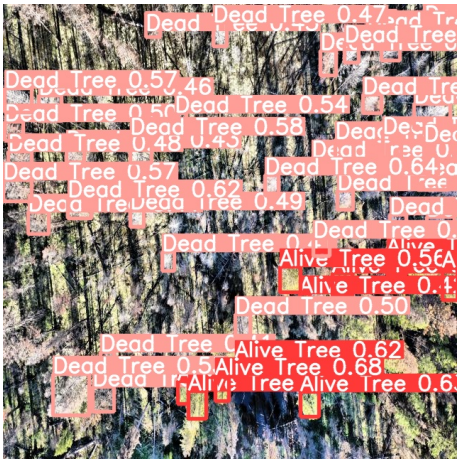

Fig. 15. Test image 2

C. Pre-processing and augmentation

It is a generally accepted notion that bigger datasets result in better Deep Learning models. However, assembling enormous datasets can be a very daunting task due to the manual effort of collecting and labeling data. Limited datasets is an especially prevalent challenge in image analysis.

In our project, we have only 483 pictures, but we need to divide them into two labels and there are only a little different between tress and background. So, the amount of this dataset is too small to get the precise result, especially for the dead trees.

Roboflow gives us a plenty of options for augmentation and pre-procressing, which mostly helped us to focus on how to improve the precision. We tried multiple methods which include: tile, saturation, exposure, isolate, adaptive equalization. And we finally decide to use adaptive equalization, because of the best results.

In the following table, we could find that the isolate method is abnormal. Isolate processing means using every annotation as a single image for training and testing, which excludes the background and thus achieves very high precision. but actually, when we apply our model to new images, we need to work on the whole image with the background, so the isolation method can be only used for training but not for detecting new images, and the precision rate will not be that high. So we decided to use adaptive equalization, which we called adjust contrast in general.

	Overall Precision	Precision of alive trees	Precision of dead trees
None	0.56	0.61	0.51
Tile with Saturation	0.578	0.62	0.527
Tile with Exposure	0.551	0.603	0.501
Only Exposure	0.573	0.595	0.551
Only Saturation	0.573	0.622	0.524
Isolate	0.997	0.997	0.997
Adaptive Equalization	0.581	0.633	0.526
Adaptive Equalization and Exposure	0.543	0.595	0.492

Fig. 16. Comparison all methods

What is contrast? At the heart of contrast is the observable difference in the condition. In the image, this means that we capture the noticeable difference in the subject. In the most basic terms, this means that there is a big difference between pixels.

Crucially, contrast does not apply a blanket filter to increase/decrease all pixels, such as 20 percent brightness. Instead, the pixels in the image are adjusted on a relative basis: the darker pixels in the entire image are ”smoothed”.

Why use contrast to pre-process images? When comparing our images of the trees between background, not only does the image on the right look more pleasant, but our neural network is also easier to understand. Recall that one basic tenant in Computer Vision (whether it’s classification, object detection, or segmentation) is edge detection. When using contrast pre-processing, edges become sharper because neighboring pixel differences are exaggerated.

In general, if a problem is considered to have a low-contrast image or part of an image with saturated contrast, it is helpful to smooth the contrast of the image by pre-processing. A common task with lower contrast than expected is working with scanned documents. In low-contrast situations, inferring weak letters for optical character recognition (OCR) can be challenging. Creating a greater contrast between the letters and the background can make the edges sharper. Note that the contrast change isn’t just about darkening the entire image: the white background is almost the same shade. So, in our task, using this method was able to increase the difference between the trees and the background to get better results.

Fig. 17. Diff between original picture with adaptive equalization

D. Detect new images and map coloring

We use a google colab notebook to detect the trees in new images and label risk levels by map coloring. First we detect the trees in bounding boxes, and then merge the boxes within a defined distance threshold because we think trees close to each other have similar fire risk. And then we color the images as green indicating low risk, the alive tree boxes as yellow indicating medium risk and dead tree boxes as red indicating high risk.

1) import packages and define paths: first we import YOLOv5 and required packages.

```
!git clone https://github.com/ultralytics/yolov5
%cd yolov5
%pip install -qr requirements.txt

import torch
import os
import random
import cv2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN

print(f"Setup complete. Using torch {torch.__version__} ({torch.cuda.get_device_properties(0).name if torch.cuda.is_available() else 'CPU'})")
```

Fig. 18. import packages

Then we define the paths needed to store different stages of images.

```
# file paths
base_path = '/content/drive/MyDrive/ML Proj RA/detect_merge_map_workflow'
images_path = os.path.join(base_path, 'images')
resized_path = os.path.join(base_path, 'resized')
visualize_path = os.path.join(base_path, 'visualize') # optional merging result visualization
out_path = os.path.join(base_path, 'out')
labels_path = '/content/yolov5/runs/detect/exp3/labels' # exp# to change
```

Fig. 19. define paths

2) detect images and save detected bounding boxes in txt file: first we read all images and resize them into 640*640, because our model is trained by 640*640 images.

```
imgs = [] # including all original size images
dim = (640, 640)

for fname in sorted(os.listdir(images_path)):
  img = cv2.imread(os.path.join(images_path, fname))
  imgs.append(img)
  img_resized = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
  img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
  cv2.imwrite(os.path.join(resized_path, f"resized_{fname}"), img_resized)
```

Fig. 20. read and resize images

Then we run detect.py with trained weights and save the detected bounding boxes into txt file with –save-txt parameter

```
!python detect.py \
        --weights '/content/drive/MyDrive/ML Proj RA/detect_merge_map_workflow/final_4_best.pt' \
        --img 640 \
        --conf 0.1 \
        --source '/content/drive/MyDrive/ML Proj RA/detect_merge_map_workflow/resized' \
        --save-txt
```

Fig. 21. detect bounding boxes

3) merge bounding boxes: first we define the merging distance threshold at 0.1 eps, and read bounding boxes information

```
eps = 0.1 # pixel distance threshold
imgs_boxes = [] # including all boxes from all images
# read boxes per image
for img_id, fname in enumerate(sorted(os.listdir(labels_path))):
  boxes = read_img_boxes(os.path.join(labels_path, fname), imgs[img_id].shape)
  imgs_boxes.append(boxes)
```

Fig. 22. read bounding boxes

Then we can visualize the bounding boxes in the image before merging

```
# OPTIONAL: visualize all boxes from all images
for img_id, img in enumerate(imgs): # per image
  for cls_id, boxes_cls in enumerate(imgs_boxes[img_id]): # per class
    for box in boxes_cls: # per box
      img = draw_box(img, cls_id, box)
  show_img(img)
```

Fig. 23. visualize bounding boxes

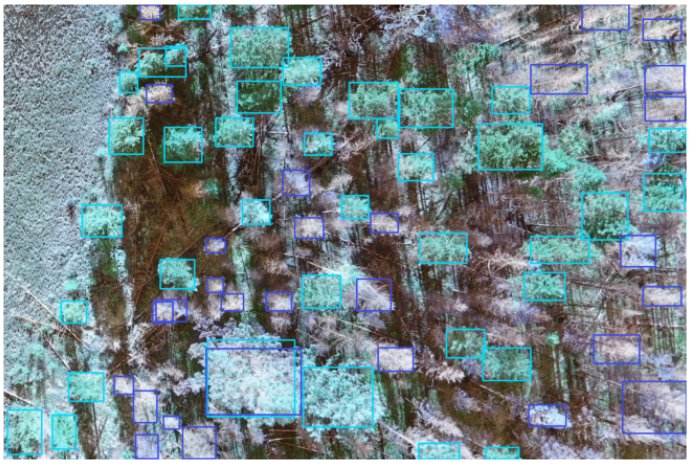

Fig. 24. bounding boxes before merging

Then we can merge the bounding boxes and visualize merging results

```
imgs_merged_boxes = [] # including all merged boxes from all images
for boxes in imgs_boxes: # per image
  new_boxes = []
  for cls_id, boxes_cls in enumerate(boxes): # per class
    # Empty class handling
    if len(boxes_cls) == 0:
      new_boxes.append([])
      continue

    # DBSCAN - algorithm to merge boxes
    """
    eps: distance in pixels to merge (0.1 to merge intersected boxes)
    min_samples: minimum 2 boxes can merge together
    metric: distance function (defined `rect_distance`)
    """
    result = DBSCAN(eps=eps, min_samples=2, metric=rect_distance).fit(boxes_cls)
    # print(result.labels_)

    # Create ONE new box per merged boxes
    new_boxes_cls = []
    clusters = {cls: [] for cls in set(result.labels_) if cls != -1} # group boxes with same class in dictionary
    for id, cls in enumerate(result.labels_): # add all single (class==-1) boxes
      if cls == -1:
        new_boxes_cls.append(boxes_cls[id])
      else:
        clusters[cls].append(boxes_cls[id])
    for key, val in clusters.items():
      box_res = val[0]
      for box in val:
        box_res = merge_boxes(box_res, box)
      new_boxes_cls.append(box_res)
    new_boxes.append(new_boxes_cls)
  imgs_merged_boxes.append(new_boxes) # append all merged boxes per image
```

Fig. 25. merge bounding boxes

```
# visualize merging result
fnames = os.listdir(images_path)
for img_id, (img, fname) in enumerate(zip(imgs, fnames)): # per image
  for cls_id, boxes_cls in enumerate(imgs_merged_boxes[img_id]): # per class
    for box in boxes_cls: # per box
      img = draw_box(img, cls_id, box)
  show_img(img)
  cv2.imwrite(os.path.join(visualize_path, f"merge_vis_{fname}"), img)
```

Fig. 26. visualize merging results

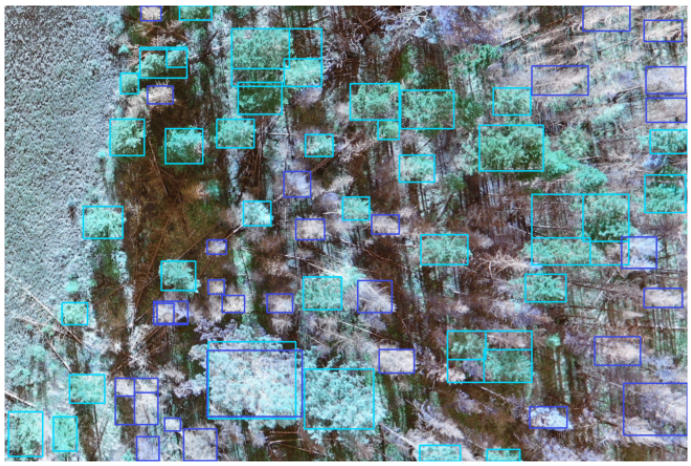

Fig. 27. merged bounding boxes

4) map coloring: first we color all the images as green indicating low risk

```
color = (70, 220, 70)
alpha = 0.4
imgs2 = [] # including all original size images

# read and map green color on all images for efficiency
for fname in sorted(os.listdir(images_path)):
  img = cv2.imread(os.path.join(images_path, fname))
  img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
  # map green color
  img_layer = cv2.rectangle(img.copy(), (0, 0), (img.shape[1] - 1, img.shape[0] - 1), color, thickness=-1, lineType=cv2.LINE_AA)
  img = cv2.addWeighted(img_layer, alpha, img.copy(), 1 - alpha, 0)
  imgs2.append(img)
```

Fig. 28. color all images green

Then we color the bounding boxes in the images. Alive tree boxes are colored yellow indicating medium risk and dead tree boxes are colored red indicating high risk

```
def draw_map(img, cls, box):
  """
  Overlay ONE filled bounding box on images with highlight option
  """
  colors = [(255, 216, 0), (231, 76, 60)] # [yellow, red]
  # colors = [(0, 216, 255), (60, 76, 231)] # [yellow, red]
  alpha = 0.5
  img_layer = cv2.rectangle(img.copy(), (box[0], box[1]), (box[2], box[3]), colors[cls], -1)
  img = cv2.addWeighted(img_layer, alpha, img.copy(), 1 - alpha, 0)
  return img
```

Fig. 29. draw map function

```
fnames = os.listdir(images_path)
for img_id, (img, fname) in enumerate(zip(imgs2, fnames)): # per image
  for cls_id, boxes_cls in enumerate(imgs_merged_boxes[img_id]): # per class
    for box in boxes_cls: # per box
      img = draw_map(img, cls_id, box)
  show_img(img)
  cv2.imwrite(os.path.join(out_path, f"map_vis_{fname}"), cv2.cvtColor(img, cv2.COLOR_RGB2BGR))
```

Fig. 30. color boxes

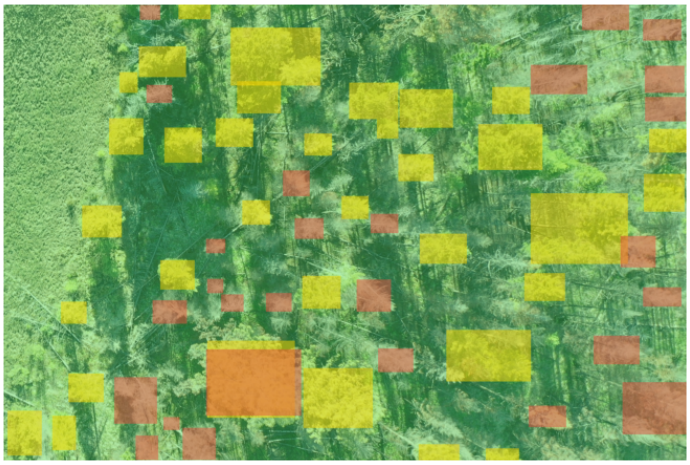

Fig. 31. map coloring result

IV. Results

Overall we achieved 58.1 percent precision as shown in precision figure and 53.3 percent mAP (mean average precision) for the detection of all the classes as in the mAP figure.

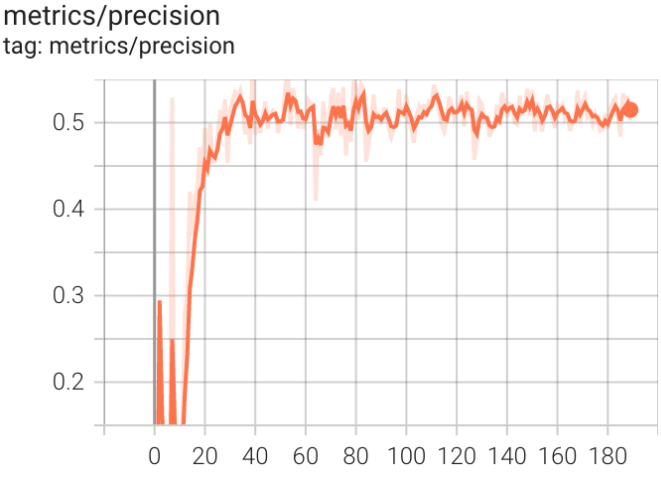

Fig. 32. Precision

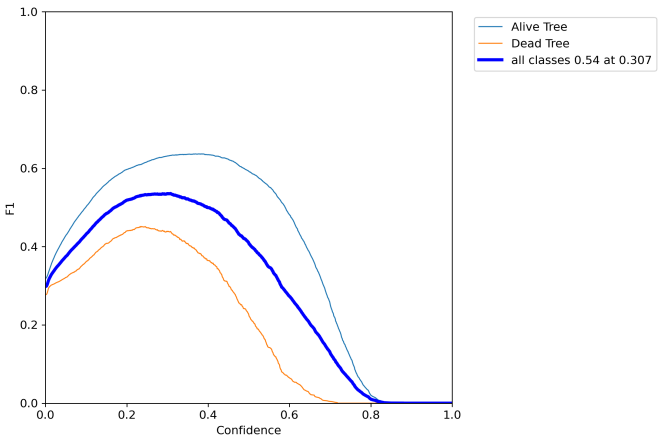

Fig. 33. mAP of all class

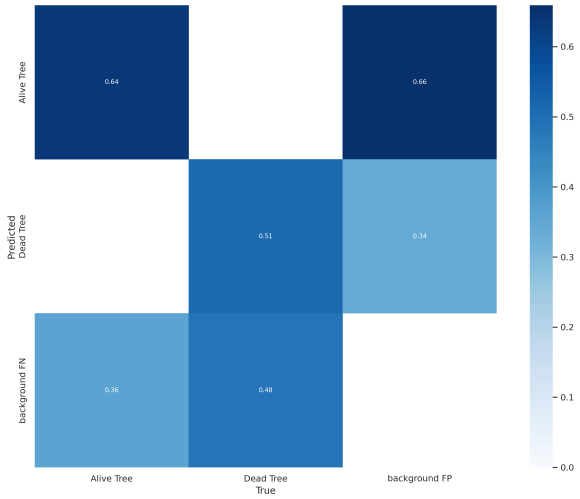

Fig. 34. Confusion Matrix

Individually, we achieved 63.3 percent precision of detection for alive tree with 61.5 mAP. For dead trees, we achieved 52.6 percent precision of detection and 38.9 mAP.

The reason that the precision of dead trees was lower than alive trees is that these dead trees were almost white that were too similar like the snow background, and it was very hard to classify, which the Confusion Matrix figure could show us well.

Although, that is a irregular confusion matrix, it could also show many deeply important information. As we can see X-axis is the true option that is noted by us, Y-axis is the output by YOLO. The first line means the precision of the areas which we noted as alive tree. Background FN means this area is not background, but YOLO regarded it as background wrongly. We could find there are 36 percent which YOLO mistook this areas for background, which should belong to alive tree. And the mistake of background and dead trees reached up to almost 50 percent, that is very high.

Same as the third line, Background FP means this areas should be regarded as the background, but YOLO has 66 percent to mistakenly believe as alive trees and has 34 percent regarded it as dead trees.

In a words, YOLO has the high probability to mistakenly regard the background as the alive tree, and may also mistakenly regard the dead tree as the background. Both of reasons I said could have an effect on this bad result at the meantime. But I also have a good news, that is we did the nice job at the classification between the alive tree and dead tree.

V. Conclusion and Future Work

In this paper we have used computer vision tools and machine learning algorithm to detect tree types and estimate risk level of forest to help predict and prevent wildfires. Raw drone images are annotated on Roboflow, and pre-processing methods such as Tile and Isolate Objects, and augmentation methods such as Saturation and Exposure are used to improve model performance.

We used precision and mAP to evaluate the model performance, and have achieved overall 58.1 percent precision and 53.3 percent mAP. Considering the complexity of the raw images, the results are already pretty good. Map coloring algorithm are used to label the risk levels in output images.

For future work, we can try more pre-processing and augmentation methods to see whether model performance and results can be further improved. We can also improve map coloring algorithm to assess the risk probability of each bounding box and create gradient map coloring.

References

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[2] R. Xu, H. Lin, K. Lu, L. Cao, and Y. Liu, "A forest fire detection system based on ensemble learning," *Forests*, vol. 12, no. 2, 2021. [Online]. Available: https://www.mdpi.com/1999-4907/12/2/217

[3] D. Thuan, "Do thuan evolution of yolo algorithm and yolov5: The state-of-the-art object detection algorithm evolution of yolo algorithm and yolov5: The state-of-the-art object detection algorithm," 2021.