Eyes on Eats: From Image to Formula

Contribution Table

Team Member	Checkpoint [Contribution %]	Final [Contribution %]
bhanucha	50	50
charviku	50	50

Report Overview

This document provides an overview of our project, Eyes on Eats, including the advancements achieved to date. For those reviewing this at the stage of the checkpoint, it will include the methods employed for the collection and preparation of the data, the challenges encountered during annotation, and strategies that may be used for automation in the process of annotation, and possibly how the confirmation of the integrity of the structure of the data is carried out. After preparation of the data, formatting to YOLO format, and split, this data is used for training two versions of the YOLOv7 model.

For the final part of the submission we delve into the fine-tuning process of the YOLOv7 model, as well as the introduction of YOLOv9 along with its variants, version 1 and version 2. Subsequently, we conduct a comparative analysis of the model outcomes. Additionally, we explore the transformer model, examining the data processing and tokenization procedures, followed by its training using the BART model, along with the resultant findings. Further insights into these aspects will be discussed in detail.

About the Project

"Eyes on Eats," aims to address the challenge many individuals face when they are unsure of what to cook with the ingredients available. This uncertainty often leads to wasted time contemplating meal options or even unnecessary spending on ordering food. "Eyes on Eats" offers a solution by employing advanced deep learning techniques in object detection and text generation. By analyzing images of ingredients, the system generates personalized recipes tailored to the user's available ingredients. This innovative approach not only streamlines the cooking process but also
encourages culinary creativity. With "Eyes on Eats," users can confidently embark on their culinary journey without the stress of meal planning, ultimately saving time and potentially reducing unnecessary expenses.

Objectives:

- Develop a robust object detection model capable of accurately identifying various ingredients depicted in images.
- Implement an efficient text generation model to seamlessly translate detected ingredients into personalized recipe recommendations.
- Ensure the scalability and adaptability of the system to accommodate a wide range of ingredients and recipes.

Datasets

We need two types of data, for this project one going to be the image data of the ingredients to train the model on the object detection and we need textual data second we need the to train the other model to generate the recipe according to the depicted ingredients.

Object Detection Data

We explored various ingredients datasets around the internet and they lack the requirements we need, and very short for training the complex model, so we manually do the web scraping using the Bing image downloader, but we noticed there's inconsistency in the image formats and the bounded by the limitation of which we can only download one class at a time. So we modified it for our requirements and scrape 100 classes of images utilizing this.

🔗 Access the tool here!

Here's the list of ingredients we are going to using the tool:

all_purpose_flour	almonds	apple	apricot	asparagus
avocado	bacon	banana	barley	basil
basmati_rice	beans	beef	beets	bell_pepper
berries	biscuits	blackberries	black_pepper	blueberries
bread	bread_crumbs	bread_flour	broccoli	brownie_mix
brown_rice	butter	cabbage	cake	cardamom
carrot	cashews	cauliflower	celery	cereal
cheese	cherries	chicken	chickpeas	chocolate
chocolate_chips	chocolate_syrup	cilantro	cinnamon	clove
cocoa_powder	coconut	cookies	corn	cucumber
dates	eggplant	eggs	fish	garlic
ginger	grapes	honey	jalapeno	kidney_beans
lemon	mango	marshmallows	milk	mint
muffins	mushroom	noodles	nuts	oats
okra	olive	onion	orange	oreo_cookies
pasta	pear	pepper	pineapple	pistachios
pork	potato	pumpkin	radishes	raisins
red_chilies	rice	rosemary	salmon	salt

shrimp	spinach	strawberries	sugar	sweet_potato
tomato	vanilla_ice_cream	walnuts	watermelon	yogurt

After the scraping the data is stored in a directory in a structured format, where everything is every category is subdivided into separate directories and each containing the images of the respective category. For now we collected the image classification data. But we need the object detection data. Before that we need to clean and verify the collected data.

Correcting the initial data

The class name in the above table has underscore but we cant use such names to scrape the data form the web, as that can lead to less accurate results, so we to scrape the keywords are provided in such a way that it wont effect the data search. As you can see form the given example.

```
queries = [ "baking powder", "basil", "cereal", "cheese", "chicken"]

for query in queries:
    if len(sys.argv) == 3:
        filter = sys.argv[2]
    else:
        filter = ""

    downloader.download(
        query,
        limit=50,
        output_dir="dataset_dem",
        adult_filter_off=True,
        force_replace=False,
        timeout=120,
        filter=filter,
        verbose=True,
    )
```

The above process in the scraping lead to creating the directory names to “baking powder” which can lead to various inconsistences in the further processes. So we created these steps to ensure consistency:

- **Convert Spaces in Directory Names to Underscores**: Rename directories to replace spaces with underscores to avoid inconsistencies. For example, rename "all purpose

flour" to "all_purpose_flour".

```
Renamed 'all purpose flour' to 'all_purpose_flour'
Renamed 'basmati rice' to 'basmati_rice'
Renamed 'bell pepper' to 'bell_pepper'
Renamed 'black pepper' to 'black_pepper'
```

- **Verify Folder Names Against Class List**: Ensure all folder names match exactly with the classes listed in a "Final_classes.txt" file. This step checks for both missing directories and extra directories not listed in the class list.

```
All classes in 'Final_classes.txt' have corresponding directories in
the dataset.
No extra directories in the dataset that are not listed in 'Final_cla
sses.txt'.
```

- **Remove Non-JPG Files**: Execute a script to traverse the dataset directories and remove any files that are not in .jpg format. This is crucial for maintaining consistency in the file format across the dataset.

```
def remove_non_jpg_images(dataset_dir):
    removed_files = []
    for root, dirs, files in os.walk(dataset_dir):
        for file in files:
            # Check if the file extension is not .jpg
            if not file.lower().endswith('.jpg'):
                file_path = os.path.join(root, file)
                os.remove(file_path)  # Remove the non-JPG file
                removed_files.append(file_path)
    return removed_files


dataset_dir = r'C:\Users\Kiyo\Desktop\DL\Project\image_data\initial_d
ata'
removed_files = remove_non_jpg_images(dataset_dir)

if removed_files:
    print(f"Removed {len(removed_files)} non-JPG files:")
```

```
    for file in removed_files:
        print(file)
else:
    print("No non-JPG files found in the dataset.")
```

- **Check for Class Image Count**: Ensure that each class directory contains exactly 50 images. If a class has more than 50 images, randomly remove the excess images to limit each class to 50.

```
all_purpose_flour: 50 images
almonds: 50 images
apple: 50 images
apricot: 50 images
asparagus: 50 images
avocado: 50 images
bacon: 50 images
..
..
```

- **Augment Images for Underrepresented Classes**: For classes with fewer than 50 images, perform image augmentation to increase the total to 50 images per class. This ensures uniformity in the number of images across all classes.

```
Completed augmentation for class 'all_purpose_flour'.
Completed augmentation for class 'almonds'.
Completed augmentation for class 'apple'.
Completed augmentation for class 'apricot'.
Completed augmentation for class 'asparagus'.
Completed augmentation for class 'avocado'.
Completed augmentation for class 'bacon'.
```

Annotating the object detection data

The dataset is ready, it consists of 100 classes present in the dataset, and each class contains 50 samples. But this is in the image classification format, which means to satisfy the stated objective of the project, The model is to be provided with pictures of one ingredient at a time and, after all the ingredients are collected by the user, with the help of the encoded vectors, it generates recipes through text generation. Which was very

inconvenient, but also annotating the image data is very inconvenient. Then we discovered Grounding Dino. A zero-shot object detection model.

Step 1: Check GPU Availability

Use `!nvidia-smi` to check if a GPU is available for faster processing.

Step 2: Set Home Directory

Define a `HOME` constant to manage datasets, images, and models easily:

```
import os
HOME = os.getcwd()
print(HOME)
```

Step 3: Install Grounding DINO

Clone the Grounding DINO repository, switch to a specific feature branch (if necessary), and install the dependencies:

```
%cd {HOME}
!git clone https://github.com/IDEA-Research/GroundingDINO.git
%cd {HOME}/GroundingDINO

# we use latest Grounding DINO model API that is not official yet
!git checkout feature/more_compact_inference_api

!pip install -q -e .
!pip install -q roboflow dataclasses-json onemetric
```

Step 4: Additional Dependencies & Verify CUDA and PyTorch

Ensure CUDA and PyTorch are correctly installed and compatible:

```
import torch
!nvcc --version
TORCH_VERSION = ".".join(torch.__version__.split(".")[:2])
CUDA_VERSION = torch.__version__.split("+")[-1]
print("torch: ", TORCH_VERSION, "; cuda: ", CUDA_VERSION)

import roboflow
import supervision
```

```
print(
    "roboflow:", roboflow.__version__,
    "; supervision:", supervision.__version__
)
```

```
# confirm that configuration file exist

import os

CONFIG_PATH = os.path.join(HOME, "GroundingDINO/groundingdino/config/
GroundingDINO_SwinT_OGC.py")
print(CONFIG_PATH, "; exist:", os.path.isfile(CONFIG_PATH))
```

Step 5: Download Configuration and Weights

Ensure the configuration file exists within the cloned repository and download the model weights:

```
# download weights file

%cd {HOME}
!mkdir {HOME}/weights
%cd {HOME}/weights

!wget -q https://github.com/IDEA-Research/GroundingDINO/releases/down
load/v0.1.0-alpha/groundingdino_swint_ogc.pth

# confirm that weights file exist

import os

WEIGHTS_PATH = os.path.join(HOME, "weights", "groundingdino_swint_og
c.pth")
print(WEIGHTS_PATH, "; exist:", os.path.isfile(WEIGHTS_PATH))
```

Step 6: Download and Prepare Your Dataset

If your dataset is zipped in your drive, unzip it to a local directory:

```
import zipfile

# Path to the zip file
zip_file_path = "/content/drive/MyDrive/....[your file path]"

# Directory to extract the contents of the zip file
extract_dir = "/content/data"

# Unzip the file
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall(extract_dir)

print("Extraction complete.")
```

Step 7: Load the Grounding DINO Model

Load the model using the configuration and weights path:

```
%cd {HOME}/GroundingDINO

from groundingdino.util.inference import Model

model = Model(model_config_path=CONFIG_PATH, model_checkpoint_path=WE
IGHTS_PATH)
```

Step 8: Annotate Dataset and Save to PASCAL voc

Use the model to annotate images. You can run inference in different modes like `caption`, `classes`, or `enhanced classes` depending on your needs. After inference, use the detections and labels to annotate images using your preferred method or the provided utility functions.

Automate the annotation process for your entire dataset by iterating over your images, running the model to detect objects, and saving both the annotated images and their PASCAL VOC XML files.

```
import os
import cv2
import xml.etree.ElementTree as ET
from groundingdino.util.inference import Model
from tqdm import tqdm
```

```
# Define the home directory and the path to the dataset
HOME = "/content"
DATASET_DIR = os.path.join(HOME, "data", "ingredients_images_datase
t")

# Load the Grounding DINO model
MODEL_CONFIG_PATH = os.path.join(HOME, "GroundingDINO", "groundingdin
o", "config", "GroundingDINO_SwinT_OGC.py")
WEIGHTS_PATH = os.path.join(HOME, "weights", "groundingdino_swint_og
c.pth")
model = Model(model_config_path=MODEL_CONFIG_PATH, model_checkpoint_p
ath=WEIGHTS_PATH)

# Load class labels from the file
LABELS_FILE_PATH = "[ txt file path containing your images labels one
per line]"
with open(LABELS_FILE_PATH, "r") as f:
    CLASSES = [line.strip() for line in f.readlines()]

# Define annotation thresholds
BOX_THRESHOLD = 0.35
TEXT_THRESHOLD = 0.25

# Function to enhance class names
def enhance_class_name(class_names):
    return [f"all {class_name}s" for class_name in class_names]

# Function to create Pascal VOC format XML annotation
def create_pascal_voc_xml(image_filename, image_shape, boxes, label
s):
    annotation = ET.Element("annotation")

    folder = ET.SubElement(annotation, "folder")
    folder.text = "ingredient_annotations"  # Folder name for annotat
ions

    filename = ET.SubElement(annotation, "filename")
    filename.text = image_filename
```

```
    source = ET.SubElement(annotation, "source")
    database = ET.SubElement(source, "database")
    database.text = "Unknown"

    size = ET.SubElement(annotation, "size")
    width = ET.SubElement(size, "width")
    height = ET.SubElement(size, "height")
    depth = ET.SubElement(size, "depth")

    width.text = str(image_shape[1])
    height.text = str(image_shape[0])
    depth.text = str(image_shape[2])

    segmented = ET.SubElement(annotation, "segmented")
    segmented.text = "0"

    for box, label in zip(boxes, labels):
        object = ET.SubElement(annotation, "object")
        name = ET.SubElement(object, "name")
        pose = ET.SubElement(object, "pose")
        truncated = ET.SubElement(object, "truncated")
        difficult = ET.SubElement(object, "difficult")
        bndbox = ET.SubElement(object, "bndbox")
        xmin = ET.SubElement(bndbox, "xmin")
        ymin = ET.SubElement(bndbox, "ymin")
        xmax = ET.SubElement(bndbox, "xmax")
        ymax = ET.SubElement(bndbox, "ymax")

        name.text = label
        pose.text = "Unspecified"
        truncated.text = "0"
        difficult.text = "0"
        xmin.text = str(int(box[0]))
        ymin.text = str(int(box[1]))
        xmax.text = str(int(box[2]))
        ymax.text = str(int(box[3]))

    # Format the XML for better readability
    xml_string = ET.tostring(annotation, encoding="unicode")
```

```
    return xml_string

# Function to annotate images in a directory and save annotated image
s in Pascal VOC format
def annotate_images_in_directory(directory):
    for class_name in CLASSES:
        class_dir = os.path.join(directory, class_name)
        annotated_dir = os.path.join(directory, f"{class_name}_annota
ted")
        os.makedirs(annotated_dir, exist_ok=True)

        print("Processing images in directory:", class_dir)
        if os.path.isdir(class_dir):
            for image_name in tqdm(os.listdir(class_dir)):
                image_path = os.path.join(class_dir, image_name)
                image = cv2.imread(image_path)
                if image is None:
                    print("Failed to load image:", image_path)
                    continue

                detections = model.predict_with_classes(
                    image=image,
                    classes=enhance_class_name([class_name]),
                    box_threshold=BOX_THRESHOLD,
                    text_threshold=TEXT_THRESHOLD
                )
                # Drop potential detections with phrase not part of C
LASSES set
                detections = detections[detections.class_id != None]
                # Drop potential detections with area close to area o
f the whole image
                detections = detections[(detections.area / (image.sha
pe[0] * image.shape[1])) < 0.9]
                # Drop potential double detections
                detections = detections.with_nms()

                # Create the Pascal VOC XML annotation for this image
                xml_annotation = create_pascal_voc_xml(image_filename
=image_name, image_shape=image.shape, boxes=detections.xyxy, labels=
[class_name])
```

```
                # Save the Pascal VOC XML annotation to a file
                xml_filename = os.path.join(annotated_dir, f"{os.pat
h.splitext(image_name)[0]}.xml")
                with open(xml_filename, "w") as xml_file:
                    xml_file.write(xml_annotation)

                # Save the annotated image
                annotated_image_path = os.path.join(annotated_dir, im
age_name)
                cv2.imwrite(annotated_image_path, image)

# Annotate images in the dataset directory
annotate_images_in_directory(DATASET_DIR)
```

Now, wr’re use it to automate the process of annotating the dataset in Pascal VOC format. Which will be in the following format, An image that belong to some class and xml file for that respective image.

```
<annotation>
<folder>ingredient_annotations</folder>
<filename>Image_1.jpg</filename>
<source>
<database>Unknown</database>
</source>
<size>
<width>1920</width>
<height>1280</height>
<depth>3</depth>
</size>
<segmented>0</segmented>
<object>
<name>almonds</name>
<pose>Unspecified</pose>
<truncated>0</truncated>
<difficult>0</difficult>
<bndbox>
<xmin>252</xmin>
<ymin>650</ymin>
<xmax>803</xmax>
<ymax>920</ymax>
</bndbox>
</object>
</annotation>
```

Verifying the Annotated Data

- Check if all the images are annotated by checking if the image file has the respective xml file, if not we remove and manually create a new annotated sample.

```
def check_dataset_integrity(dataset_directory):
    for class_name in os.listdir(dataset_directory):
        class_path = os.path.join(dataset_directory, class_name)
        if os.path.isdir(class_path):
            jpg_files = set()
            xml_files = set()
```

```
            other_files = set()

            # Collect file names for each extension
            for file_name in os.listdir(class_path):
                if file_name.endswith('.jpg'):
                    jpg_files.add(os.path.splitext(file_name)[0])
                elif file_name.endswith('.xml'):
                    xml_files.add(os.path.splitext(file_name)[0])
                else:
                    other_files.add(file_name)

            # Check for discrepancies
            missing_xmls = jpg_files - xml_files
            missing_jpgs = xml_files - jpg_files
            is_perfect = len(missing_xmls) == 0 and len(missing_jpgs)
== 0 and len(other_files) == 0

            # Report
            print(f"Class '{class_name}':", "Perfect" if is_perfect e
lse "Discrepancies Found")
            if missing_xmls:
                print(f"  Missing XML files for: {', '.join(sorted(mi
ssing_xmls))}")
            if missing_jpgs:
                print(f"  Missing JPG files for: {', '.join(sorted(mi
ssing_jpgs))}")
            if other_files:
                print(f"  Non-JPG/XML files: {', '.join(sorted(other_
files))}")
        else:
            print(f"'{class_name}' is not a directory. Skipping.")

# Specify the path to the dataset directory
dataset_directory = r'C:\Users\Kiyo\Desktop\DL\Project\image_data\ini
tial_data_annotated'
check_dataset_integrity(dataset_directory)
```

```
# Output Sample
Class 'all_purpose_flour_annotated': Perfect
```

```
Class 'almonds_annotated': Perfect
Class 'apple_annotated': Perfect
Class 'apricot_annotated': Perfect
Class 'asparagus_annotated': Perfect
```

- Renamed all the directories containing samples, as you can see the dir names changed after annoateion, they are named as <some class name>_annotated. Now we remove that so that ensuring that the class name in the text file matches with the dir names.
- Again after these changes we checked if all the images have the respective annotations and dir names matches with the class list text file.

This completes our dataset preparation which is the major part in our project and took lot of time to reach the consistency through various trail and error approaches we created and perfected the dataset for the object detection training.

Text Generation Data

The RecipeNLG dataset, available in RecipeNLG_dataset.csv, encompasses 2,231,142 cooking recipes sourced from RecipeNLG. This extensive dataset, totaling 2.14 GB, contains crucial recipe details such as titles, ingredients, directions, links, sources, and Named Entity Recognition (NER) labels. With label distribution categorized into various ranges and a vast array of unique values, the dataset showcases a diverse and comprehensive collection of cooking recipes. This dataset serves as a valuable resource for training and evaluating models in a multitude of natural language processing tasks, particularly in the context of generating cooking-related text.

🔗 Access Here!

Sample

Title	Ingredients	Link	Directions	NER
No-Bake Nut Cookies	["1 c. firmly packed brown sugar", "1/2 c. evaporated milk", "1/2 tsp. vanilla", "1/2 c. broken nuts...	www.cookbooks.com/Recipe-Details.aspx?id=44874	["In a heavy 2-quart saucepan, mix brown sugar, nuts, evaporated milk and butter or margarine.", "St...	["brown sugar", "milk", "vanilla", "nuts", "butter", "bite size shredded rice biscuits"]

For training the BART transformer model, we need to prepare the tokenized data. First, the dataset is extracted using the unzip command to access the recipe data. Next, we imported libraries such as pandas, transformers, tqdm, numpy, and TensorFlow are imported.

```
!unzip '/user/bhanucha/recipe_data.zip' -d '/user/bhanucha/data'
import pandas as pd
from transformers import BartTokenizer
from tqdm import tqdm
import numpy as np
import tensorflow as tf
from transformers import TFBartForConditionalGeneration
import numpy as np
```

The BART tokenizer is initialized from the pretrained BART model, and if the tokenizer lacks a padding token, it is added. The dataset is then loaded into a pandas DataFrame.

```
model_checkpoint = "facebook/bart-base"
tokenizer = BartTokenizer.from_pretrained(model_checkpoint)
if tokenizer.pad_token is None:
    tokenizer.add_special_tokens({'pad_token': tokenizer.eos_token})


data = pd.read_csv('/user/bhanucha/data/dataset/full_dataset.csv')
```

Subsequently, the ingredients and directions from each recipe are concatenated into text strings and tokenized using the BART tokenizer. The tokenized data is then processed to ensure consistency in length and format, with the tokenized inputs saved for training.

```
texts = ["Ingredients: " + row['ingredients'] + " Directions: " + row
['directions'] for _, row in data.iterrows()]

tokenized_inputs = []
for texts_text in tqdm(texts, desc="Tokenizing Data"):
    tokenized_input = tokenizer(
        texts_text,
        padding="max_length",
        truncation=True,
```

```
        max_length=512,
        return_tensors="np"
    )
    tokenized_inputs.append(tokenized_input['input_ids'])


train_data = np.concatenate(tokenized_inputs, axis=0)
np.save('/user/bhanucha/train_data.npy', train_data)
```

Building & Training the Model

Model for Object Detection

The dataset, stored in ZIP files, was extracted and processed to convert annotations from the VOC format to YOLO format. This preprocessing step included calculating bounding box coordinates relative to image dimensions and saving them alongside class IDs in text files. The dataset was then split into training, validation, and testing sets with ratios of 80%, 10%, and 10%, respectively.

```
Training images: 3988
Validation images: 499
Test images: 499
```

Yolo_v7_Version_1

- **Batch Size**: 32
- **Image Dimensions**: 320×320
- **Epochs**: 100
- **Hyperparameters**: Default settings from YOLOv7's configuration
- **Evaluation**: Due to some limitations not tested on the test data. But anyway version 2 outperformed the version 1.

Yolo_v7_Version_2

- **Batch Size**: 64
- **Image Dimensions**: 640×640

- **Epochs**: 50
- **Hyperparameters**: Default settings from YOLOv7's configuration. Increased batch size and image resolution to explore improvements in detection accuracy and model robustness.
- **Evaluation**: Conducted tests with two IoU thresholds, 0.5 and 0.25, to assess model performance under different criteria for considering detections as true positives.

Yolo_v7_Version_3

- **Batch Size:** 64
- **Image Size:** 640×640
- **Epochs:** 100
- **Workers:** 4
- **Optimizer:** SGD (stripped after training)
- **Learning Rate:** Based on the configuration file
- **Loss Functions:** Box, Objectness, and Class Losses

YOLO_v9 Version_1

- **Batch Size:** 32
- **Image Size:** 640×640
- **Epochs:** 100
- **Workers:** 4
- **Optimizer:** SGD (stripped after training)
- **Hypothesis File:** `hyp.scratch-high.yaml`
- **Loss Functions:** Box, Objectness, and Class Losses

YOLO_v9 Version_2

- **Batch Size:** 16
- **Image Size:** 320×320
- **Epochs:** 300
- **Workers:** 4
- **Optimizer:** SGD (stripped after training)

- **Hypothesis File:** `hyp.scratch-high.yaml`
- **Loss Functions:** Box, Objectness, and Class Losses

Training Outcomes

This evaluation covers versions of the YOLOv7 object detection model trained on a dataset of various food items, assessing their performance based on precision, recall, and mean Average Precision (mAP) at an IoU threshold of 0.5.

YOLO V7 Version 1 Evaluation

- **Initial Observations**: The model started with low precision (P=0.355) and recall (R=0.0124), along with a very low mAP@0.5 (0.00473), indicating a significant portion of the detected objects were incorrectly classified or poorly localized.
- **Progression**: Throughout training, both precision and recall steadily increased, indicating an improvement in the model's ability to correctly identify and localize objects within the images.
- **Peak Performance**: By the 99th epoch, the model achieved a precision of 0.528, a recall of 0.504, and an mAP@0.5 of 0.527. These metrics demonstrate considerable improvement and a balanced capability in detecting and classifying objects accurately.
- **Training Duration**: Completed in approximately 4.077 hours, suggesting a significant investment of computational resources for training.

YOLO V7 Version 2 Evaluation

- **Initial Observations**: Similar to Version 1, the model commenced with very low performance metrics (P=0.00089, R=0.0115, mAP@0.5=0.00104). The larger image size (640×640) used in this version did not immediately translate to better initial performance.
- **Progression**: The improvement in precision, recall, and mAP@0.5 was more rapid compared to Version 1, indicating that the adjustments in hyperparameters and the increased image resolution positively impacted learning efficiency.
- **Peak Performance**: By the end of training at epoch 49, the model reached higher precision (P=0.526), recall (R=0.49), and mAP@0.5 (0.547) than Version 1. These results suggest enhanced model accuracy and reliability in object detection tasks.
- **Training Duration**: The training was completed in 1.685 hours, significantly faster than Version 1, despite the increased computational demand of processing higher resolution images.

YOLO V7 Version 3 Evaluation

- **Initial Observations**: The model started with low performance metrics, such as a precision (P) of 0.00089, recall (R) of 0.0115, and mAP@0.5 of 0.00104. This indicated the need for further tuning and training to improve its accuracy and effectiveness.
- **Progression**: Through subsequent epochs, the model's performance gradually improved, showing a steady increase in precision, recall, and mAP@0.5. By the final epoch, the model exhibited strong consistency in its ability to detect and classify objects.
- **Peak Performance**: By the end of training at epoch 99, the model demonstrated improved performance metrics:
  - **Precision:** 0.607
  - **Recall:** 0.476
  - **mAP@.5:** 0.557
  - **mAP@.5-0.95:** 0.517
    These results indicate the model's enhanced accuracy and reliability in object detection tasks.
- **Training Duration**: The training was completed in 3.344 hours, showing efficient training even with extended epochs.

YOLO V9 Version 1:

- **Initial Observations**: The model's initial performance was low, showing metrics such as precision, recall, and mAP@0.5 values at 0, indicating room for significant improvement.
- **Progression**: Over subsequent epochs, the model showed some progress in its precision, recall, and mAP@0.5:
  - **Epoch 1:**
    - **Precision:** 0.471
    - **Recall:** 0.00542
    - **mAP@.5:** 0.000338
  - **Epoch 2:**
    - **Precision:** 0.205
    - **Recall:** 0.00924
    - **mAP@.5:** 0.0000355

- **Peak Performance**: By the end of training, at epoch 99, the model achieved better metrics:
  - **Precision:** 0.416
  - **Recall:** 0.391
  - **mAP@.5:** 0.394
  - **mAP@.5-0.95:** 0.353
- **Training Duration**: The training was completed in 3.231 hours, indicating steady progress.

YOLO V9 Version 2:

- **Initial Observations**: The model's initial performance was modest, displaying low precision (P=0.0371), recall (R=0.0139), and mAP@0.5 (0.000981).
- **Progression**: Through 300 epochs, the model gradually improved its performance:
  - **Epoch 283:**
    - **Precision:** 0.446
    - **Recall:** 0.46
    - **mAP@.5:** 0.461
  - **Epoch 299:**
    - **Precision:** 0.477
    - **Recall:** 0.438
    - **mAP@.5:** 0.461
- **Peak Performance**: By the end of training, the model demonstrated steady performance:
  - **Box Loss:** 0.6706
  - **Cls Loss:** 0.9838
  - **Dfl Loss:** 1.297
- **Training Duration**: The model's training lasted 6.611 hours, demonstrating steady training progress across extended epochs.

Test Outcomes

YOLOV7 Version 2 Test Results

1. **First Test (IoU Threshold 0.5)**
    - **Precision**: 0.501
    - **Recall**: 0.546
    - **mAP@.5**: 0.532
    - **Speed**: Achieved an inference speed of 1.6 ms per image with a batch size of 64.
2. **Second Test (IoU Threshold 0.25)**
    - **Precision**: 0.54
    - **Recall**: 0.515
    - **mAP@.5**: 0.53
    - **Speed**: Slower inference speed of 2.5 ms per image at a reduced batch size of 32.

YOLO V7 Version 3

1. **Test (IoU Threshold 0.5)**
    - **Precision**: 0.581
    - **Recall**: 0.468
    - **mAP@.5**: 0.532

YOLO V9 Version 1

1. **Test (IoU Threshold 0.5)**
    - **Precision**: 0.44
    - **Recall**: 0.371
    - **mAP@.5**: 0.404

YOLO V9 Version 2

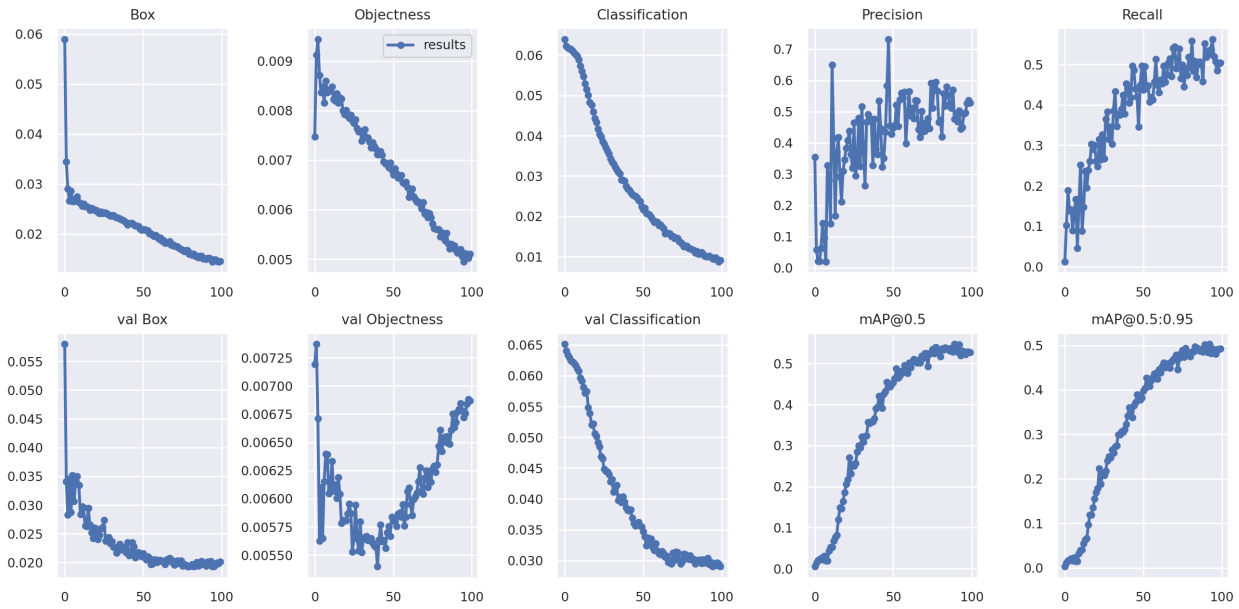
1. **Test (IoU Threshold 0.5)**
    - **Precision**: 0.267
    - **Recall**: 0.289
    - **mAP@.5**: 0.278

Outcomes

YOLO V7 Version_1

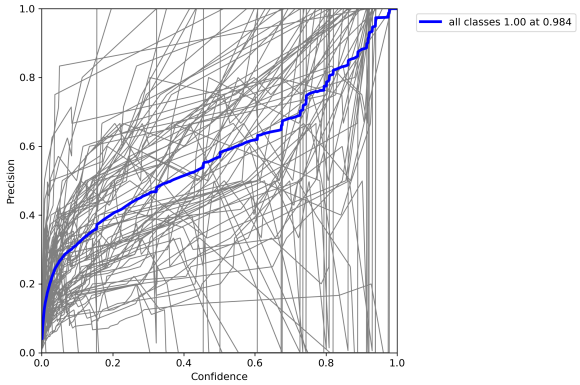
The image provided shows the plots for performance metrics for YOLOv7 Version .

1. **Box Loss**: There are two box losses, for the train data and another for the validation data, as you can see both graphs showing the loss is decreasing over time but it took some time for the model to decrease the loss
2. **Objectness Loss**: There are two box losses, for the train data and another for the validation data, which reflects confidence in the presence of an object within the box. The decreasing trend but it fluctuates in a timely manner
3. **Classification Loss**: There are two box losses, for the train data and another for the validation data, These graphs display the loss related to classifying the objects within the boxes. The consistent decrease in loss suggests that the model is getting better at correctly classifying the objects it detects.
4. **Precision**: The fluctuating line suggests variability in the model's precision across different epochs, which could be due to the model learning and adjusting its parameters. But it has higher precision at certain epoch than the final epoch
5. **Recall**: The upward trend in recall indicates the model is getting better at detecting all the relevant objects.
6. **mAP@.5**: Mean Average Precision at an IoU threshold of 0.5. The plot indicates a steady increase, meaning the model's predictions are aligning better with the ground truth as training progresses.
7. **mAP@0.5:0.95**: This is the mean Average Precision calculated over different IoU thresholds from 0.5 to 0.95 (in increments of 0.05). The steady increase shown in this plot is indicative of the model's improving accuracy across a range of strictness levels for object detection.



This image displays the objects identified by the model in a sample test batch, along with the model's confidence levels in its predictions. As evident from the image, the model is quite good at recognizing most of the items even the small ones like dates. But it failed to recognize some like cheese, cereals, chocolate. Maybe due to the form.

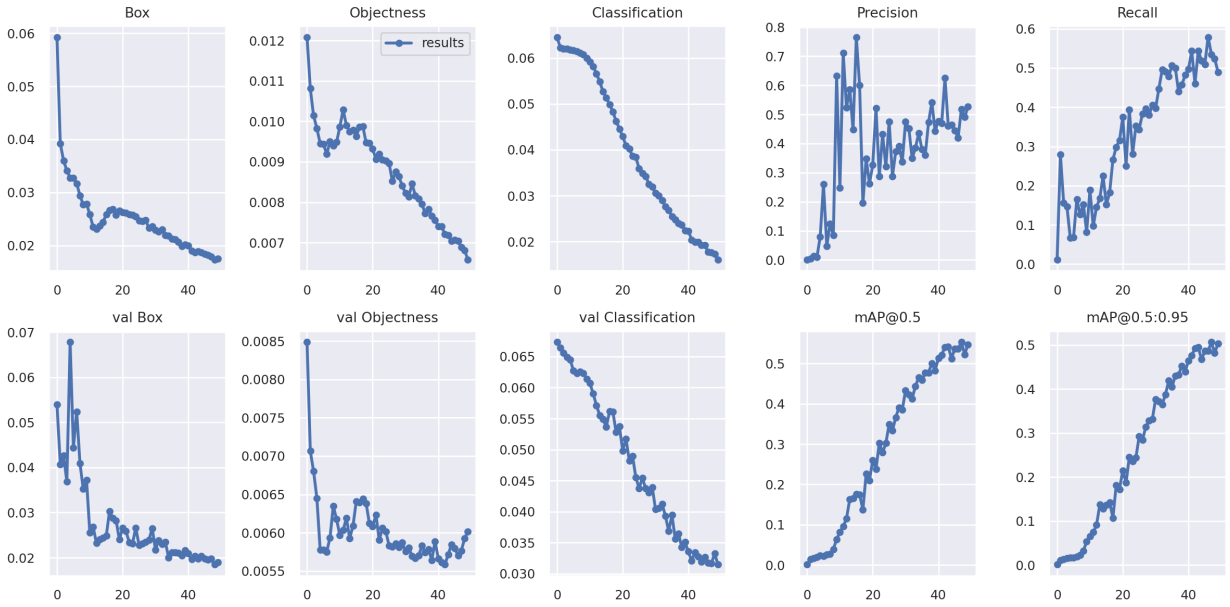
This graph displays a Precision-Confidence curve for an object detection model. The x-axis represents the confidence level of the detections, and the y-axis shows the precision of those detections. This model achieves perfect precision (1.00) at a very high confidence threshold of 0.984. This means that when the model is almost completely certain about its predictions (98.4% confident), it is accurate in identifying only the correct instances without any false positives.



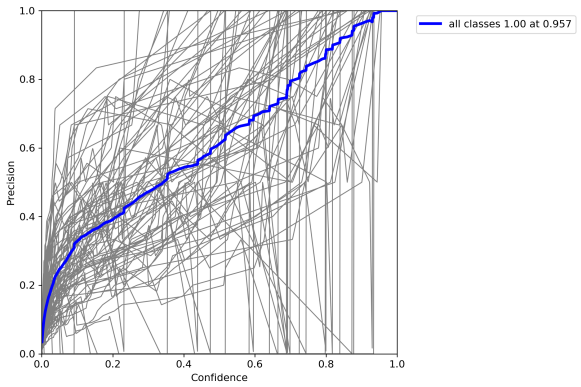
YOLO V7 Version_2

The image provided shows the plots for performance metrics for YOLOv7 Version 2.

1. **Box Loss**: There are two box losses, for the train data and another for the validation data, as you can see both graphs showing the loss is decreasing over time which indicates the model is getting better at accurately locating objects in images.
2. **Objectness Loss**: There are two box losses, for the train data and another for the validation data, which reflects confidence in the presence of an object within the box. The decreasing trend signifies that the model is improving in discerning between objects and background (i.e., becoming more confident in its predictions).
3. **Classification Loss**: There are two box losses, for the train data and another for the validation data, These graphs display the loss related to classifying the objects within the boxes. The consistent decrease in loss suggests that the model is getting better at correctly classifying the objects it detects.
4. **Precision**: The fluctuating line suggests variability in the model's precision across different epochs, which could be due to the model learning and adjusting its parameters.
5. **Recall**: The upward trend in recall indicates the model is getting better at detecting all the relevant objects.
6. **mAP@.5**: Mean Average Precision at an IoU threshold of 0.5. The plot indicates a steady increase, meaning the model's predictions are aligning better with the ground truth as training progresses.
7. **mAP@0.5:0.95**: This is the mean Average Precision calculated over different IoU thresholds from 0.5 to 0.95 (in increments of 0.05). The steady increase shown in this plot is indicative of the model's improving accuracy across a range of strictness levels for object detection.



The graph displays a Precision-Confidence curve The grey lines show precision across varying confidence levels for different object classes or predictions. The bold blue line overall precision across all classes, hitting perfect precision at 95.7% confidence.



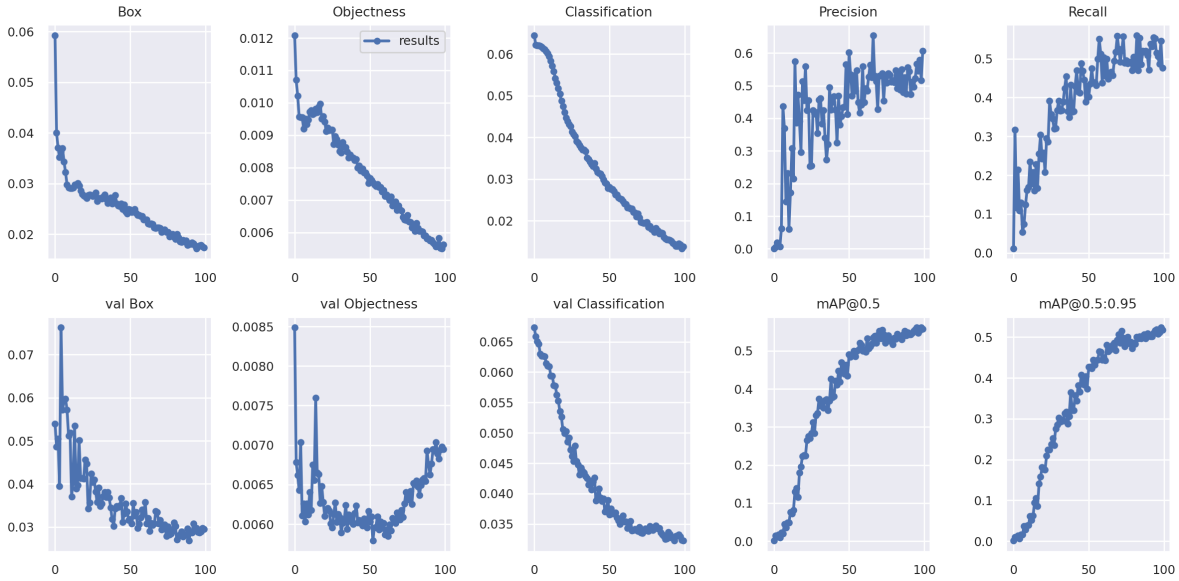
This image displays the objects identified by the model in a sample test batch, along with the model's confidence levels in its predictions. As evident from the image, the model is quite adept at recognizing certain items, while it has difficulty with others, such as rice and salt, likely due to their shape or form.

YOLO V7 Version_3

1. **Box Loss**:
    - The graph shows a sharp decrease in box loss initially, indicating quick improvements in the model's ability to precisely locate objects.
    - After the initial drop, the box loss continues to gradually decrease, stabilizing around a lower value, which suggests that the model has mostly converged in terms of predicting the correct bounding boxes for objects.
2. **Objectness Loss**:
    - This metric also shows a rapid decrease at the start, which levels off as the epochs advance.
    - The decrease in objectness loss indicates that the model is becoming increasingly better at correctly identifying the presence of an object within the bounding boxes.
3. **Classification Loss**:
    - Classification loss experiences a steep decline, reflecting improvements in the model’s ability to correctly classify the objects within the bounding boxes.
    - The steady decline and eventual plateau suggest a robust learning process where the model effectively minimizes errors in classification over epochs.
4. **Precision**:
    - The precision graph shows significant fluctuation, but overall, it trends upward, indicating increasing accuracy in the positive predictions the model makes.
    - The variability in precision could be due to the model adjusting to different object classes and their distribution in the dataset.
5. **Recall**:
    - Recall improves consistently across the training period, suggesting that the model is getting better at identifying all relevant objects in the images without missing many.
    - The consistent upward trend without much fluctuation points to the model's increased sensitivity in detecting objects as the training progresses.
6. **mAP@0.5**:
    - This metric shows a consistent upward trend, reflecting an overall improvement in both precision and recall at an Intersection over Union (IoU) threshold of 0.5.

- The continuous improvement indicates that the model is effectively learning to detect objects with at least 50% accuracy concerning their ground truth locations and classifications.

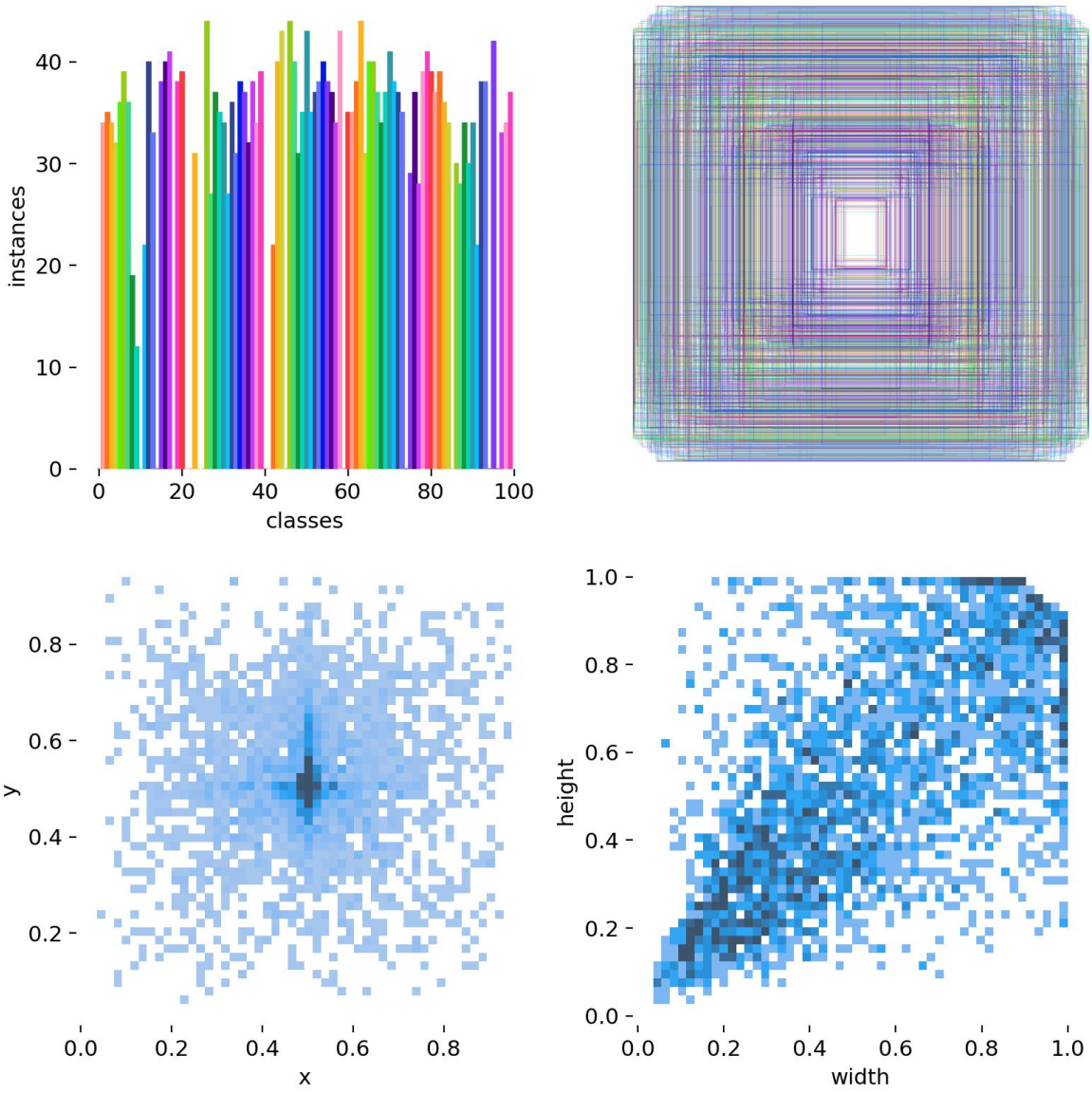
7. **mAP@0.5:0.95**:
    - Similar to mAP@0.5, mAP@0.5:0.95 also shows a smooth and consistent increase, demonstrating progressive improvements across a range of IoU thresholds from 0.5 to 0.95.
    - This improvement across thresholds signifies a robustness in the model's ability to detect objects accurately across varying degrees of overlap with the ground truth.



YOLO V9 Version_1

1. **Class Distribution**:
    - The first graph in the top row displays the instances per class across approximately 100 classes. The distribution appears relatively even, suggesting a well-balanced dataset which is crucial for training a robust object detection model.
2. **Bounding Box Locations (Grids)**:
    - The middle top graph represents the overlap of bounding boxes over an image, indicating frequent object locations in the dataset. The dense center suggests that many objects are centrally located within images.
    - The bottom-left heatmap shows the distribution of the centers (x, y coordinates) of bounding boxes. The concentration towards the middle suggests that objects are typically centered in images.

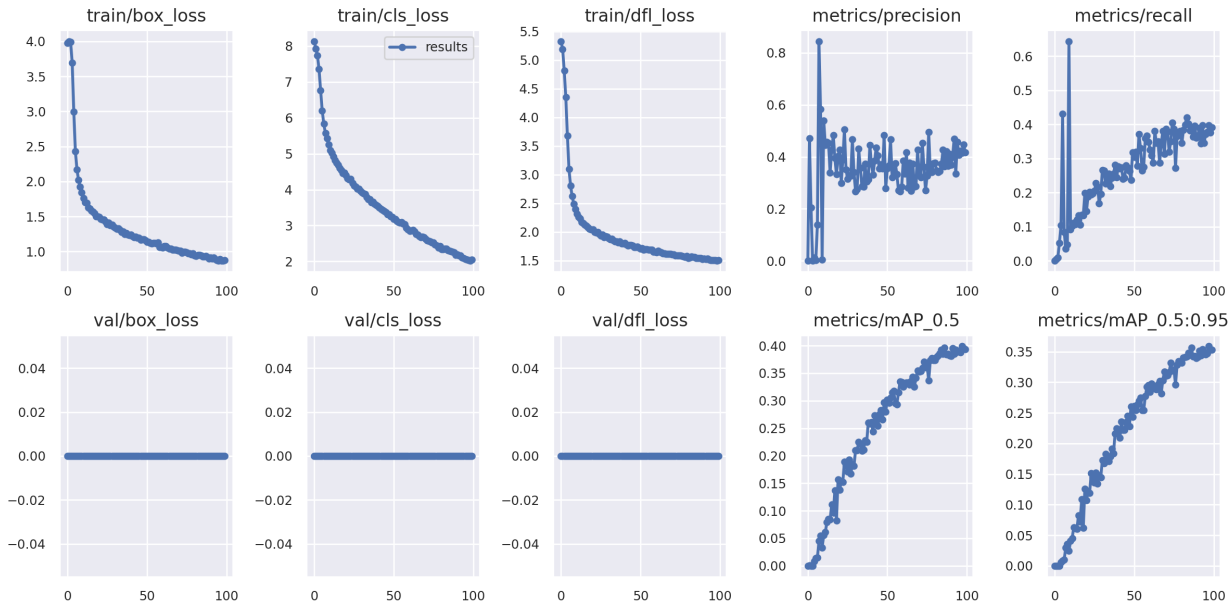
- The bottom-right heatmap represents the height and width distributions of bounding boxes, showing a trend towards smaller and wider boxes. This is important for tuning the aspect ratio considerations in the model.



3. **Loss Metrics**:
   - **Box Loss**: The top-left graph in the second image shows a sharp decline in box loss, stabilizing as epochs progress. This indicates improving accuracy in bounding box predictions.
   - **Classification Loss**: Shows a similar trend, with a rapid decline, indicating that the model is becoming better at classifying the objects within the bounding boxes accurately.

- **DFL Loss**: Represents the "Decoupled Foreground and Background Loss," showcasing a significant drop, which is beneficial for enhancing the separation between the object and background predictions.

4. **Precision and Recall**:
   - Precision is somewhat volatile but generally trends upwards, which might indicate variability in how well the model predicts across different classes.
   - Recall shows a consistent increase, suggesting the model's growing capability to detect most of the relevant objects.
5. **mAP Metrics**:
   - Both mAP@0.5 and mAP@0.5:0.95 show steady improvement. mAP@0.5, which evaluates model performance at a 50% IoU threshold, and mAP@0.5:0.95, an average over IoU thresholds from 0.5 to 0.95, both demonstrate robust enhancement, indicating increasing model accuracy and reliability across different levels of strictness in object matching.

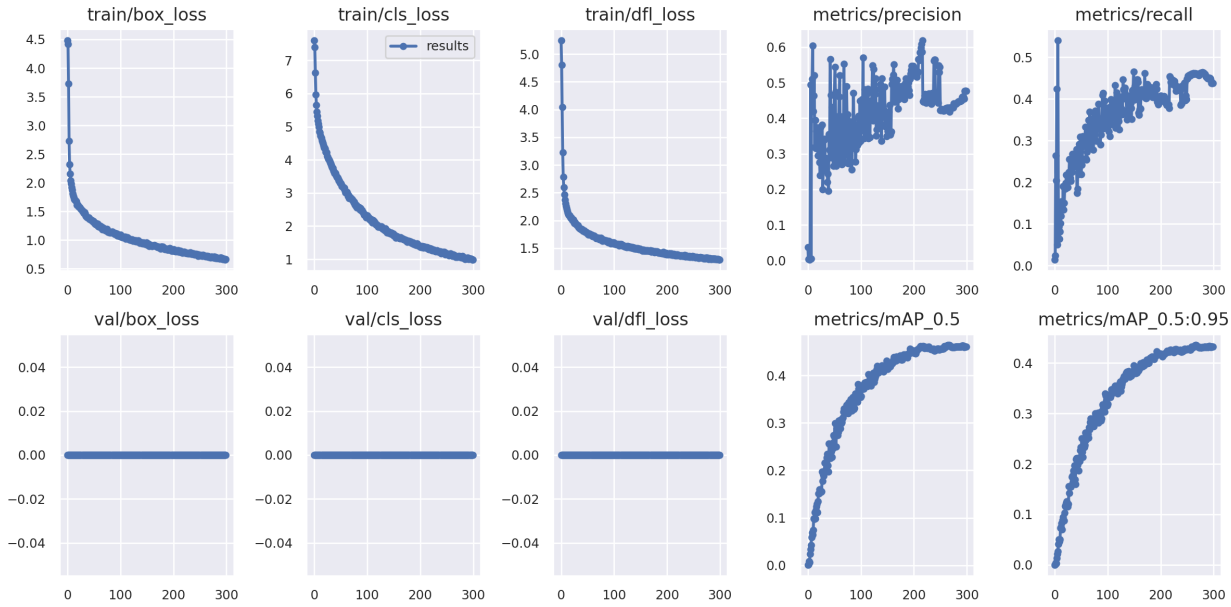


YOLO V9 Version_2

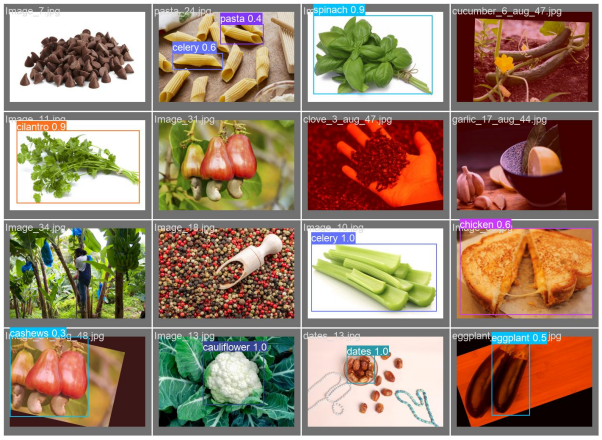
1. **Training Losses**:
   - **Box Loss**: Shows a steep decline in the initial epochs and then gradually flattens out, indicating that the model quickly improved its ability to locate objects correctly and then refined this skill over time.

- **Classification Loss**: Mirrors the trend in box loss with a sharp initial decrease, suggesting rapid improvements in correctly classifying the objects within the bounding boxes. This loss also stabilizes, indicating the model's proficiency in classification after initial learning.
- **DFL Loss**: Decreases sharply and then levels, similar to box and classification losses, showing improvements in the model's discrimination between foreground and background elements.

2. **Precision and Recall**:
    - **Precision**: Exhibits some variability but overall trends upward. The fluctuation might reflect the challenges in maintaining consistent accuracy across varying classes or scenarios within the data.
    - **Recall**: Shows a continuous increase, plateauing towards the later epochs. This consistent rise and high plateau suggest that the model is reliably identifying a high proportion of relevant objects over time.
3. **Mean Average Precision (mAP)**:
    - **mAP@0.5**: Indicates a steady increase, reflecting improvements in detection performance at the 0.5 IoU threshold. The progressive rise demonstrates the model's enhanced capability to match detected objects with ground truth at a moderate IoU threshold.
    - **mAP@0.5:0.95**: Also shows a consistent upward trend, indicating that the model performs well across a range of IoU thresholds, from lenient (0.5) to stringent (0.95). The high values achieved here reflect the model's strong ability to precisely locate and classify objects across various degrees of overlap with the ground truth.



YOLO_V9_VERSION_1



YOLO_V9_VERSION_1

Model for Text Generation

After the tokenization, we are going to train the BART transformer, to do that, accelerate and transformers libraries were installed, and essential imports were made. The objective was to leverage deep learning models, specifically BART, for recipe generation tasks. A crucial aspect was ensuring GPU availability for accelerated computation.

```
pip install accelerate
pip install transformers
from transformers import BartTokenizer, BartForConditionalGeneration
```

```
import torch
from torch.utils.data import DataLoader, TensorDataset, random_split
from transformers import Trainer, TrainingArguments
from torch.utils.data import Dataset, DataLoader, random_split
!nvidia-smi

import os
import numpy as np
import gc
os.listdir('/projects/academic/courses/cse676s24/bhanucha')
train_data_path = '/projects/academic/courses/cse676s24/bhanucha/trai
n_data_v2.npy'
train_data = np.load(train_data_path, mmap_mode='r')
print("Data type:", type(train_data))
print("Data shape:", train_data.shape)
```

The next phase involved loading and preprocessing the dataset. The RecipeNLG dataset, containing over 2 million cooking recipes, was loaded into a numpy array. The data type and shape were verified to ensure successful loading.

```
# Output
Data type: <class 'numpy.memmap'>
Data shape: (1405634, 512)
Contents of the array: [[    0 48539    35    22   134   740     4 10
523  6515  6219  4696  1297
     22   134    73   176   740     4 27805  1070  5803  1297    22
134
     73   176 26141     4 21857  1297    22   134    73   176   740
4
   3187 15092    36 26512  1253 45894    22   176   255 39596     4
9050
     50 31417 27323  1297    22   246   112    73   176   740     4 1
0970
   1836 30274  7666 31729   113 13497    35   440    12   387  5113 1
4208
  41200 38490    35    22  1121    10  2016   132    12 45252  8929 1
2560
      6  3344  6219  4696     6 15092     6 27805  1070  5803     8
9050
     50 31417 27323 45863    22  5320   853    81  4761  2859   454 1
```

```
2652
  26054    70    81   299 45863    22   387 14189     8 14351   195
728
     55     4  4624   160  2859 45863    22  5320   853    11 21857
8
  25629   131  3344   157 45863    22 36949   132 37031     6  1874
8
   3989    88   389 28255    15 19957  2225 45863    22  7939  1413
454
    933     6    59   389   728    72     2     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
...
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1     1     1     1
1
      1     1     1     1     1     1     1     1]]
```

The dataset was then converted into a custom TokenizedDataset object for efficient handling and training. A train-validation split was performed, and the dataset was

subsequently prepared for model training.

```
class TokenizedDataset(Dataset):
    def __init__(self, numpy_data):
        self.input_ids = torch.tensor(numpy_data, dtype=torch.long)
        self.attention_mask = (self.input_ids != 1).long()
        self.labels = torch.tensor(numpy_data, dtype=torch.long)

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return {
            'input_ids': self.input_ids[idx],
            'attention_mask': self.attention_mask[idx],
            'labels': self.labels[idx]
        }

dataset = TokenizedDataset(train_data)

train_size = int(0.9 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_s
ize])
```

The BART model was fine-tuned for recipe generation using a Trainer instance. Training arguments were defined to specify parameters such as output directory, number of epochs, batch size, and gradient accumulation steps. The model was trained and evaluated, with the training loss and evaluation results monitored throughout the process.

```
model_checkpoint = "facebook/bart-base"
model = BartForConditionalGeneration.from_pretrained(model_checkpoin
t)

training_args = TrainingArguments(
    output_dir='/projects/academic/courses/cse676s24/bhanucha/result
s',
    num_train_epochs=1,
    per_device_eval_batch_size=8,
    warmup_steps=500,
```

```
    weight_decay=0.01,
    logging_dir='/projects/academic/courses/cse676s24/bhanucha/logs',
    logging_steps=10000,
    evaluation_strategy="epoch",
    save_strategy="steps",
    save_steps=100000,
    save_total_limit=2,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=2,
    fp16=True,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset
)

trainer.train()
```

The model was saved after training, and its performance was evaluated based on the evaluation loss. Finally, the trained model was saved

```
model.save_pretrained('/projects/academic/courses/cse676s24/bhanucha/
saved_model')
evaluation_results = trainer.evaluate()
print(evaluation_results)
model.save_pretrained('/projects/academic/courses/cse676s24/bhanucha/
saved_model2')
```

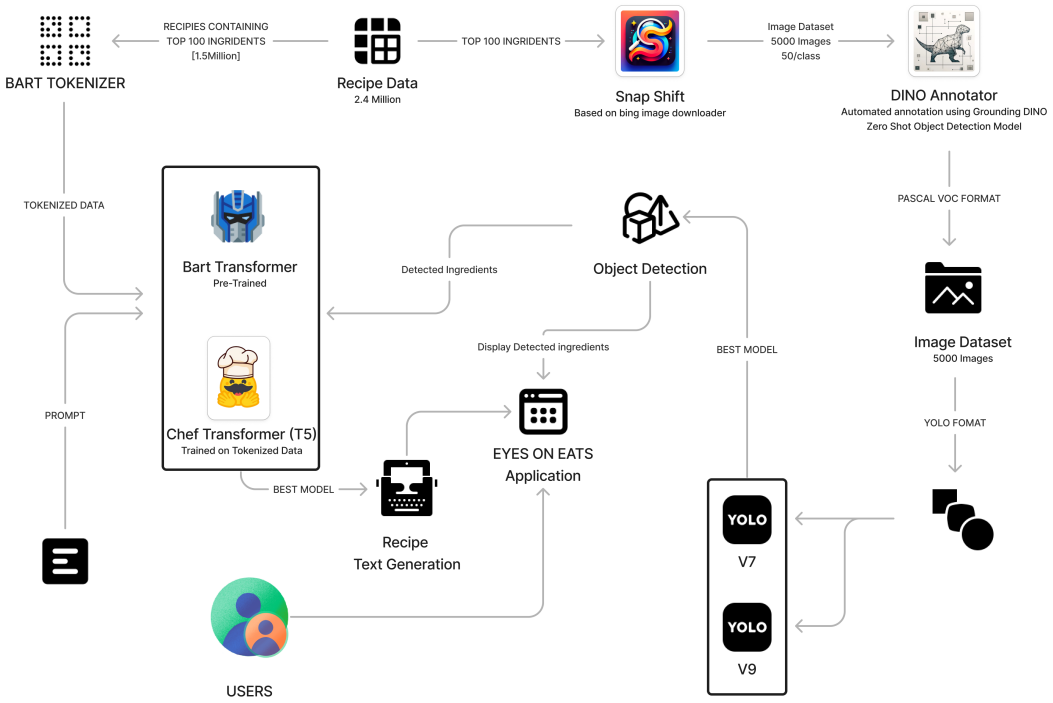
- The BART model achieved results during training, with both training and validation losses approaching zero by the end of the single epoch.
- Training loss was recorded at **0.0058**, while validation loss was even lower, at **0.000001**, indicating the model's ability to generalize to unseen data.
- The training process lasted approximately 4 hours and 40 minutes for the epoch, with a training speed of 75.147 samples per second and 9.393 steps per second.

But while giving the prompt to the model, it’s giving the output the give prompt, which indicates that the data is overfit,a nd the transformer model is unable to generate the firther test.

Final Notes

For the final project, we enhanced the YOLOv7 model and also experimented with two versions of YOLOv9. Additionally, we preprocessed and tokenized the recipe data and initially trained it using the BART transformer model. However, we observed that it was overfitting, as it merely regurgitated the inputs as outputs. Consequently, we opted to use the pre-trained ChefTransformerT5 from Hugging Face for our recipe generation model. We are including all the code snippets used so far, including those for the transformer model. Here’s a summary of our actions:

- We collected images using a customized image downloader.
- We then cleaned the data.
- We augmented the data and checked for corrections.
- Developed a custom script to automatically annotate the image data using grounding techniques.
- Using the annotated data, we trained the YOLOv7 in three variations and YOLOv9 in two variations.
- We preprocessed the recipe data.
- Tokenized this data using the BART tokenizer.
- We trained the transformer model with the tokenized data.
- After testing with a prompt, we found the transformer model was overfitting.
- Therefore, we switched to using the ChefTransformerT5, a pre-trained model available on Hugging Face.

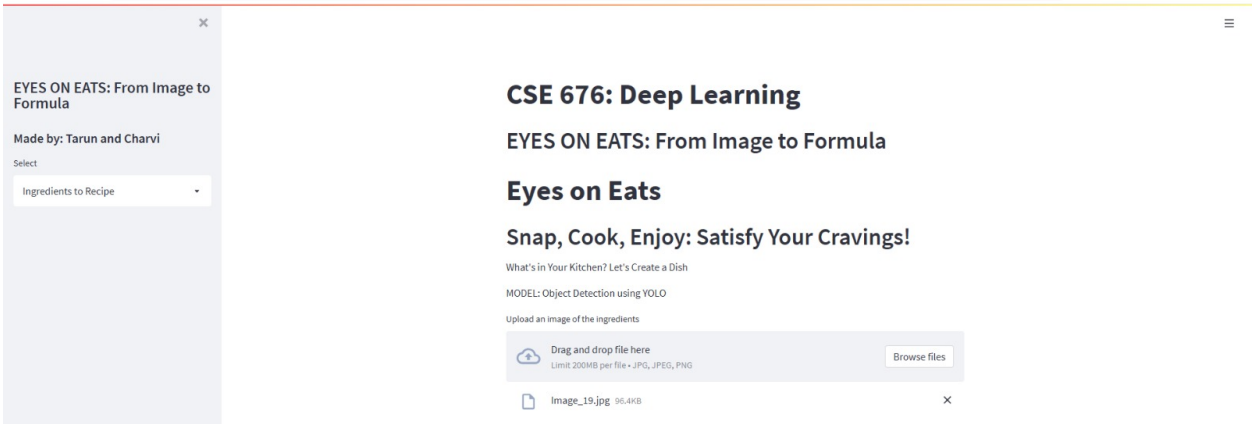


Complete Pipeline.

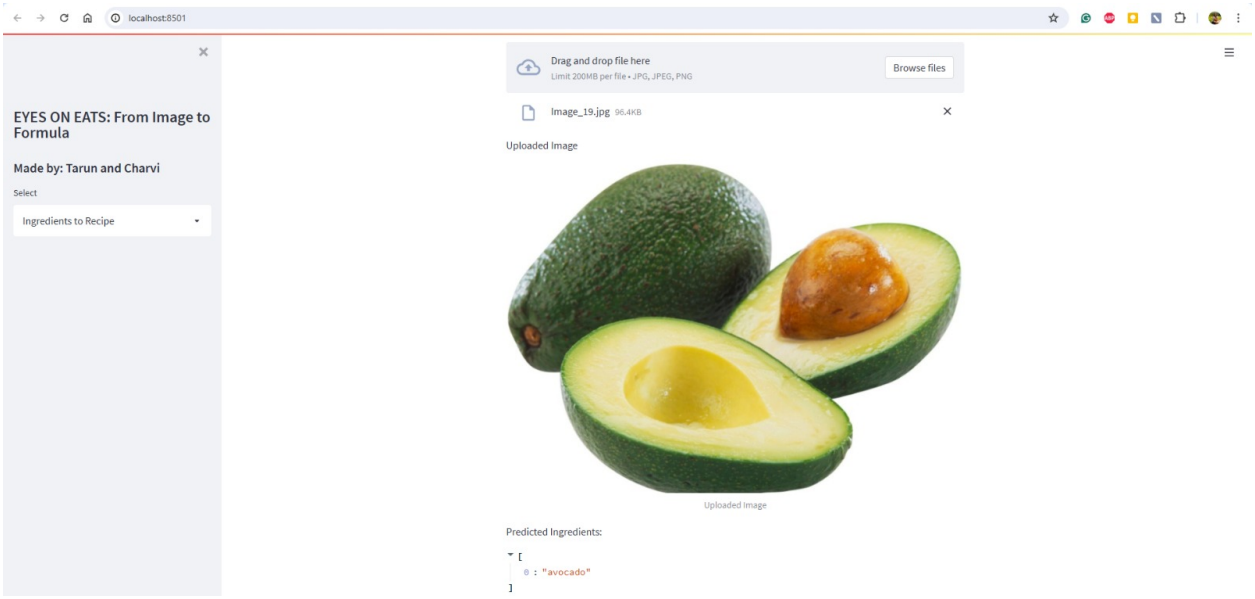
> **Note:** Eyes on eats application is a part of the demo days.

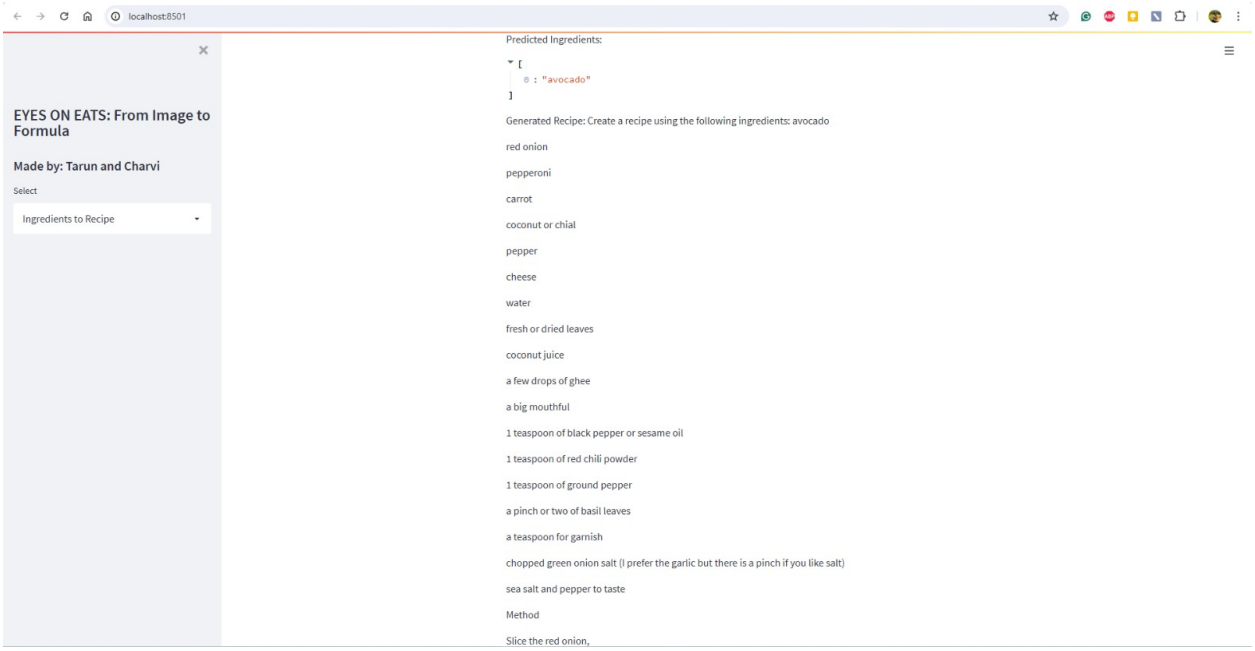
Also, we implemented an base stream lit application and deployed it on the local, the below is the screenshot for the working application. Code included in the submission.

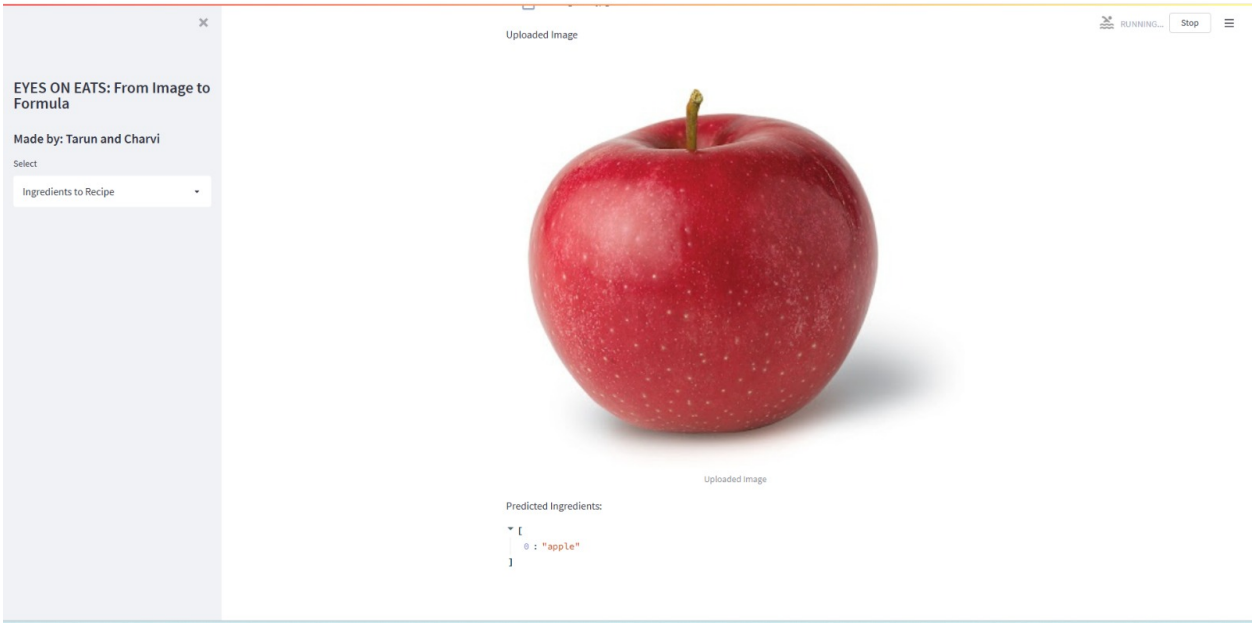
- First we need to upload the image, here we uploaded the avacado.

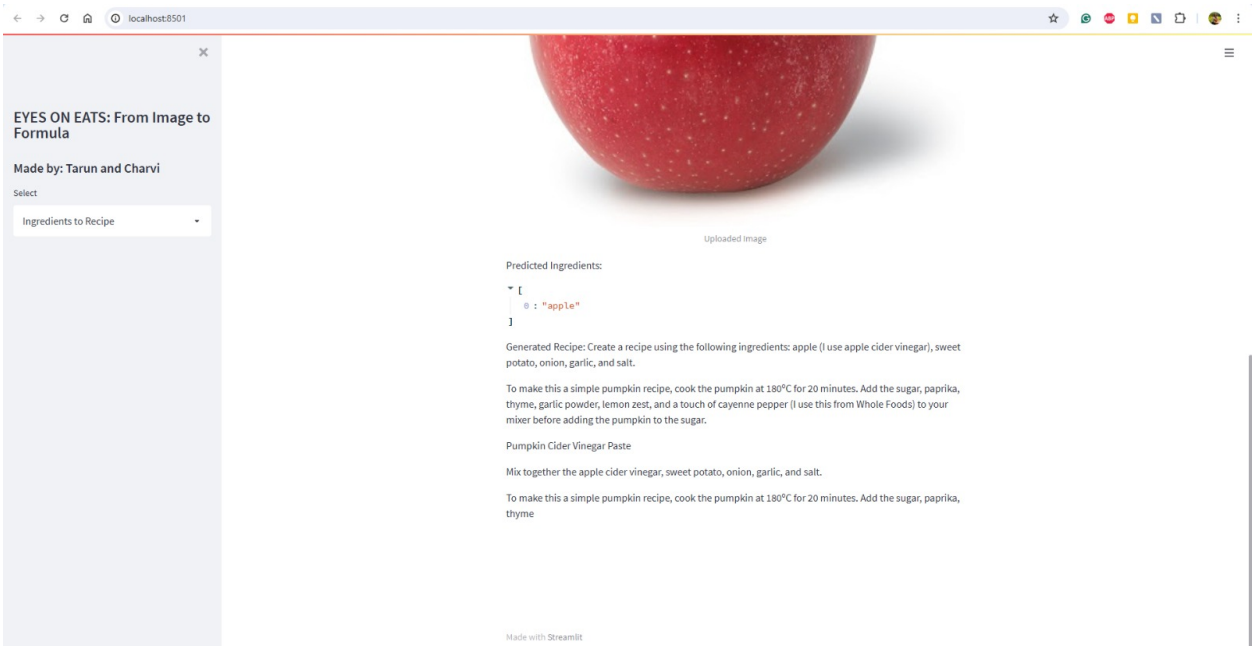


- Next it predicts the uploaded image, then generate the recipe. As it only deteted avacado in the image, it assumed the other ingredients and generate the recipe.









BONUS

Real-world deep learning application [3 points]

We created an application using the streamlit. Where we can give image of vegetables and our application produce recipes. Application will be finetuned for demo day.

Deploy the model locally [2 points]

We are deploying using streamlit. **Video is attached. Will also be presented on Demo Day**

Real-world deep learning application:

The project "Eyes on Eats" presents a real-world application of deep learning in addressing the common challenge of meal planning and cooking with available ingredients. By leveraging advanced object detection and text generation techniques, the system assists users in making informed decisions about meal preparation based on the ingredients they have on hand. This application directly addresses the practical issue of minimizing food waste, optimizing meal planning, and enhancing the overall cooking experience for individuals.

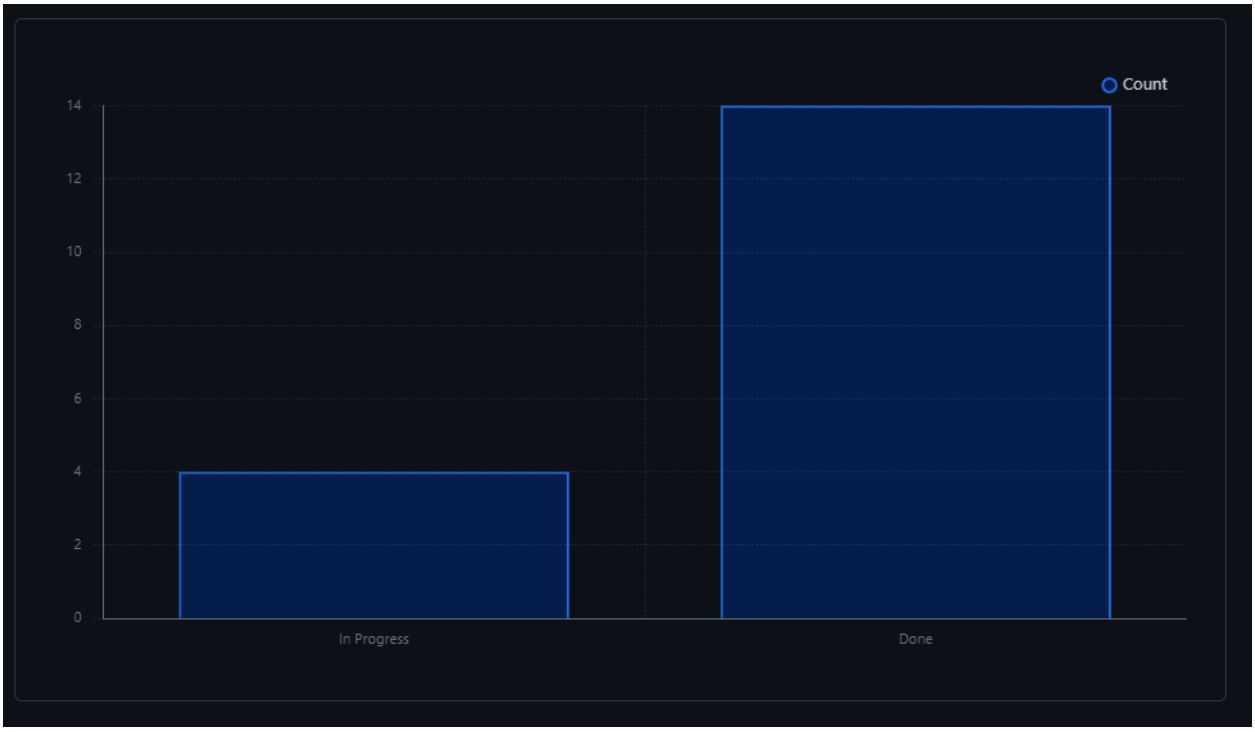
Demo Day Presentation

References

1. Anil, A. K. "GroundingDino Auto-Annotation." GitHub Repository. GitHub - GroundingDino Auto-Annotation.
2. Gaurav, G. "Bing Image Downloader." GitHub Repository. GitHub - Bing Image Downloader.
3. Anil, A. K. (Date of Publication). "Automating Image Annotation with GroundingDino." Medium Article. Medium - Automating Image Annotation with GroundingDino.
4. Hugging Face. (n.d.). *T5 Recipe Generation*. Flax Community. Retrieved from https://huggingface.co/flax-community/t5-recipe-generation

Misc.

GitHub Scrum Board [Access Here]


Count
14
12
10
8
6
4
2
0
In Progress
Done