Project Documentation

Aim: To detect Exudate in fundus image using machine learning

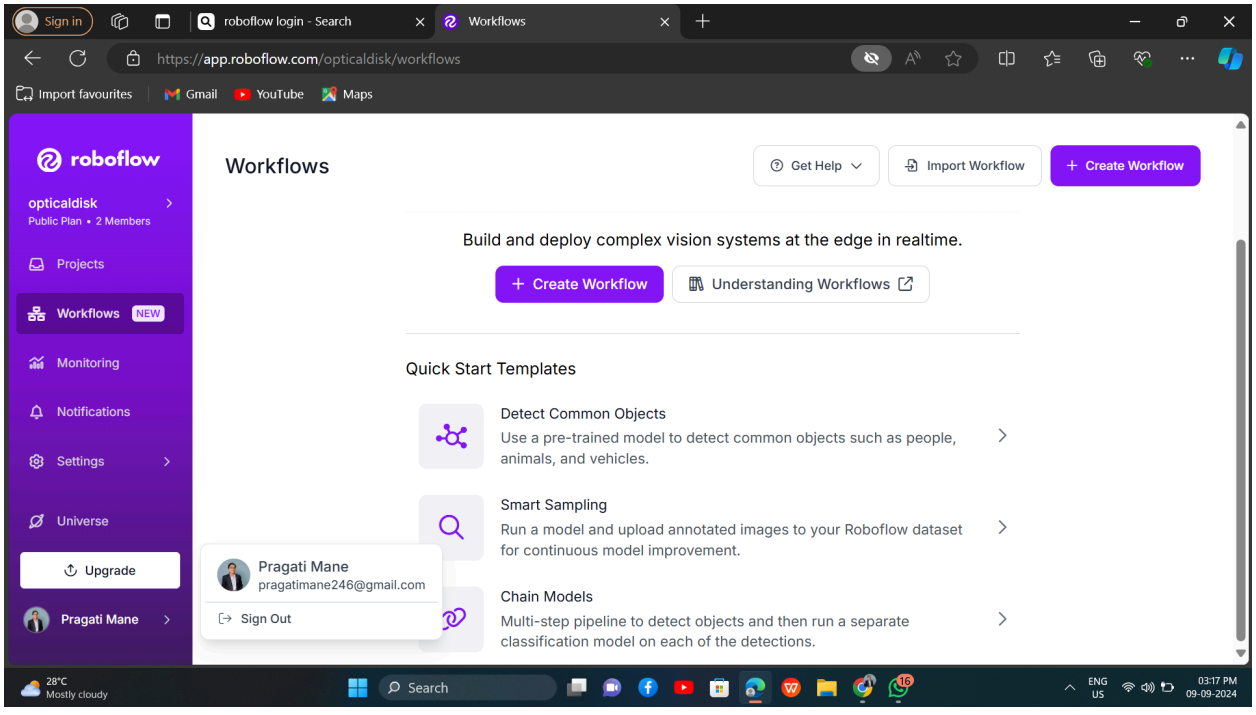
Brief of modules:

Module 1:
Aim : Detect and remove optic disc from fundus image

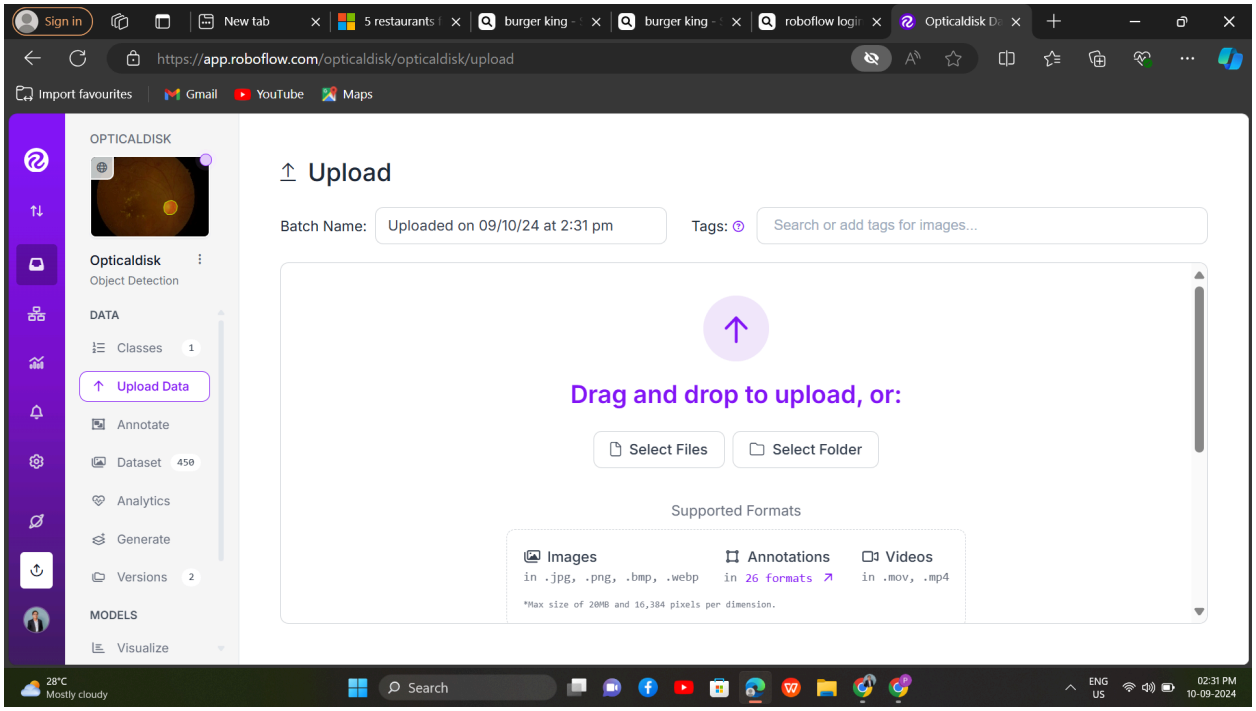
I. Algorithms can be used YOLOv10 and detectron2 to detect optic disc
Ii. Removing optic disc

Roboflow Labeling Dataset

1. Create a Roboflow Account
- Sign up or log in to [Roboflow](https://roboflow.com).
- Once logged in, you will be directed to the Roboflow dashboard.

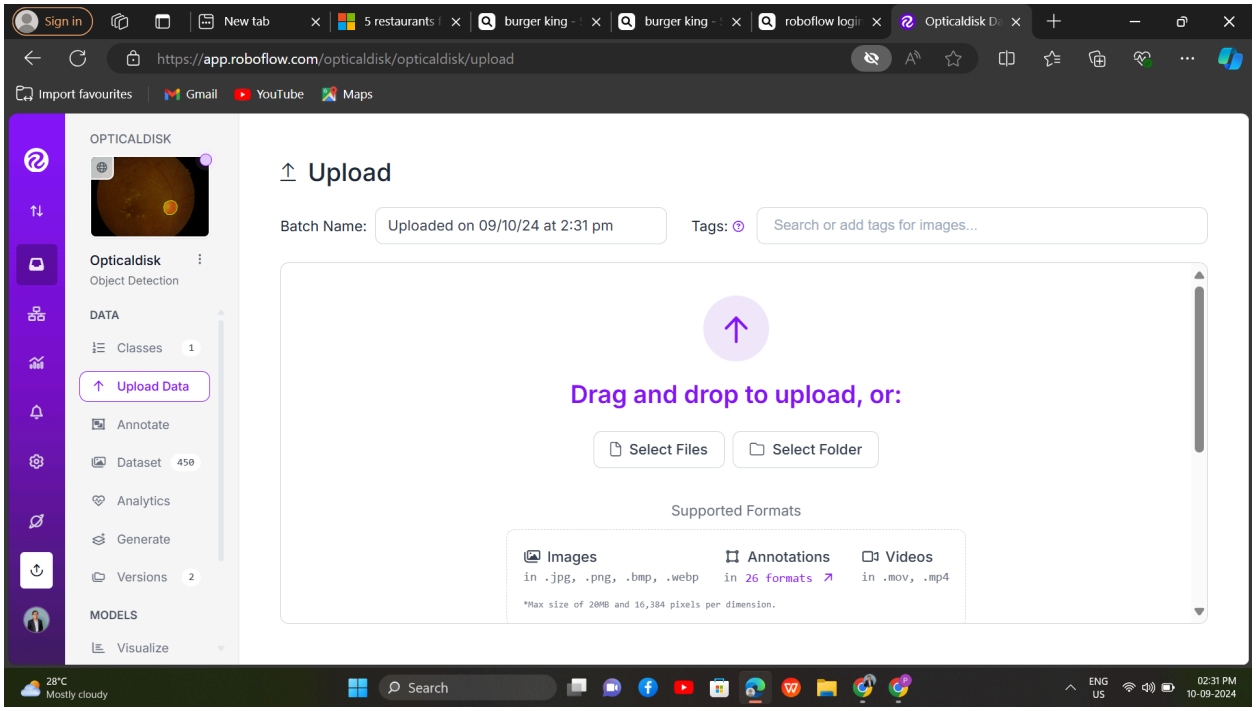


2. Create a New Project
- Click on Create New Project on the dashboard.
- Give your project a name related to diabetic retinopathy (e.g., "Diabetic Retinopathy Detection").
- Choose your task type: Object Detection (since you want to label different types of exudates).
- Select the type of annotation you want: Bounding Box for detecting hard and soft exudates.



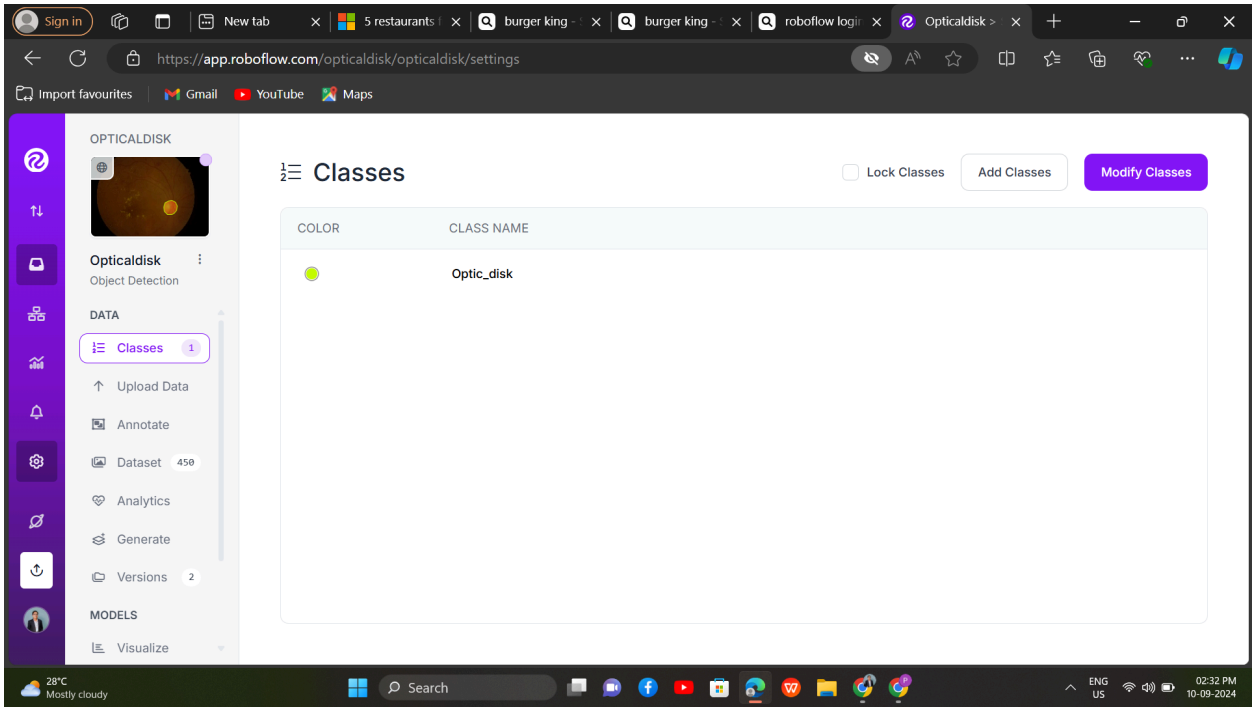
3. Upload Your Images

- After creating the project, you will be prompted to upload images.
- Drag and drop your dataset or select images from your local files. You can upload images for training, testing, and validation here.



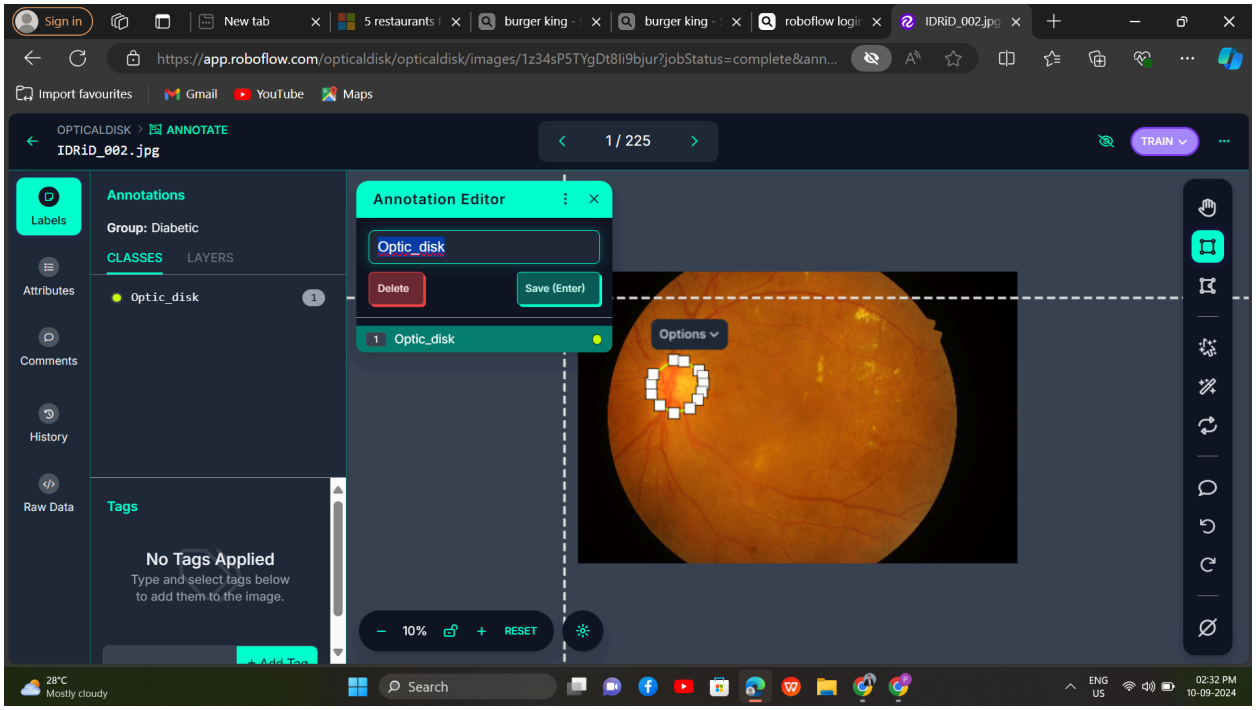
4. Add Classes for Labeling

- After uploading your images, Roboflow will ask you to define your classes.
- Add your classes: Hard Exudates, Soft Exudates, and No Exudates .

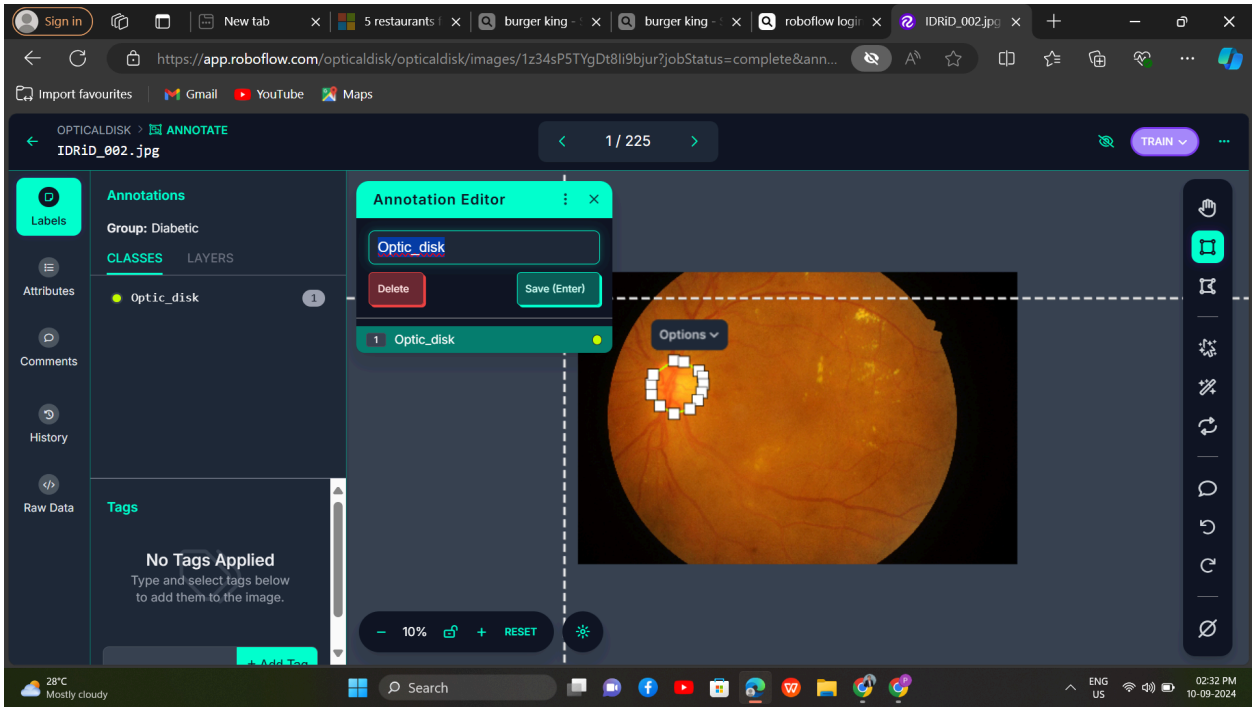


5. Annotate (Label) Your Images

- After your images are uploaded, start annotating them by drawing bounding boxes around the exudates.
- Label each region by selecting the appropriate class: Hard Exudates, Soft Exudates, or No Exudates.
- You can zoom in to accurately label the regions.

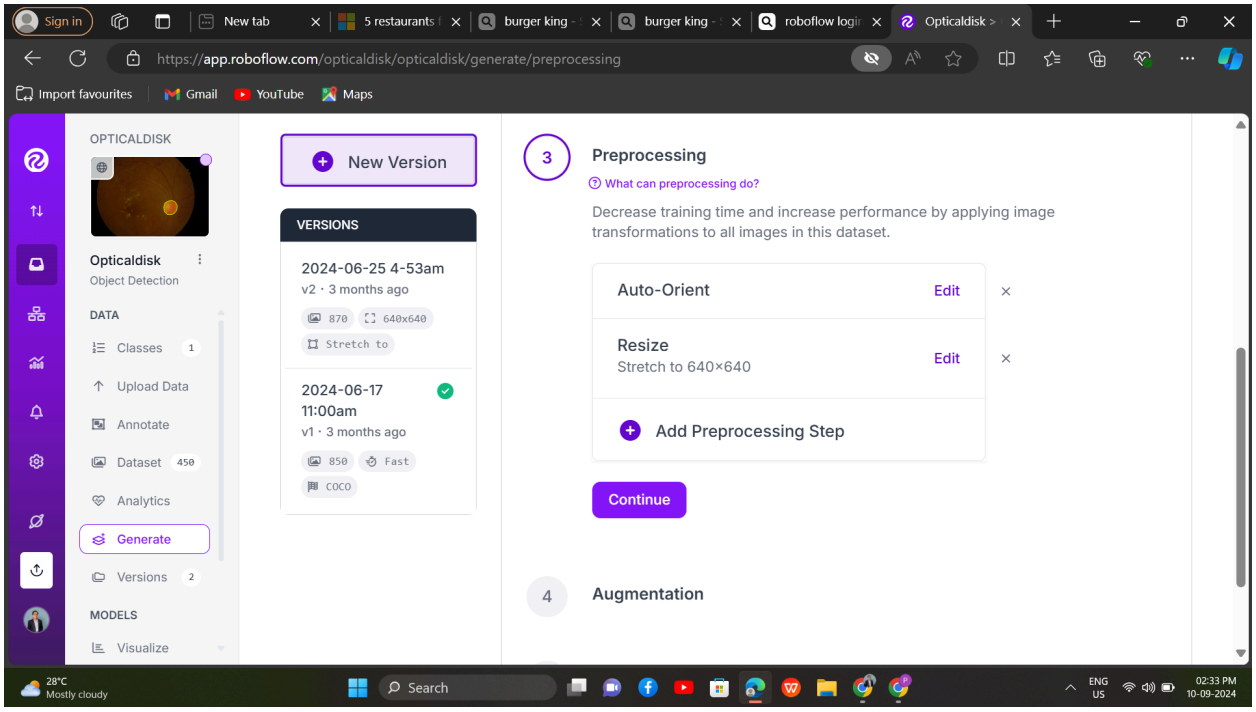


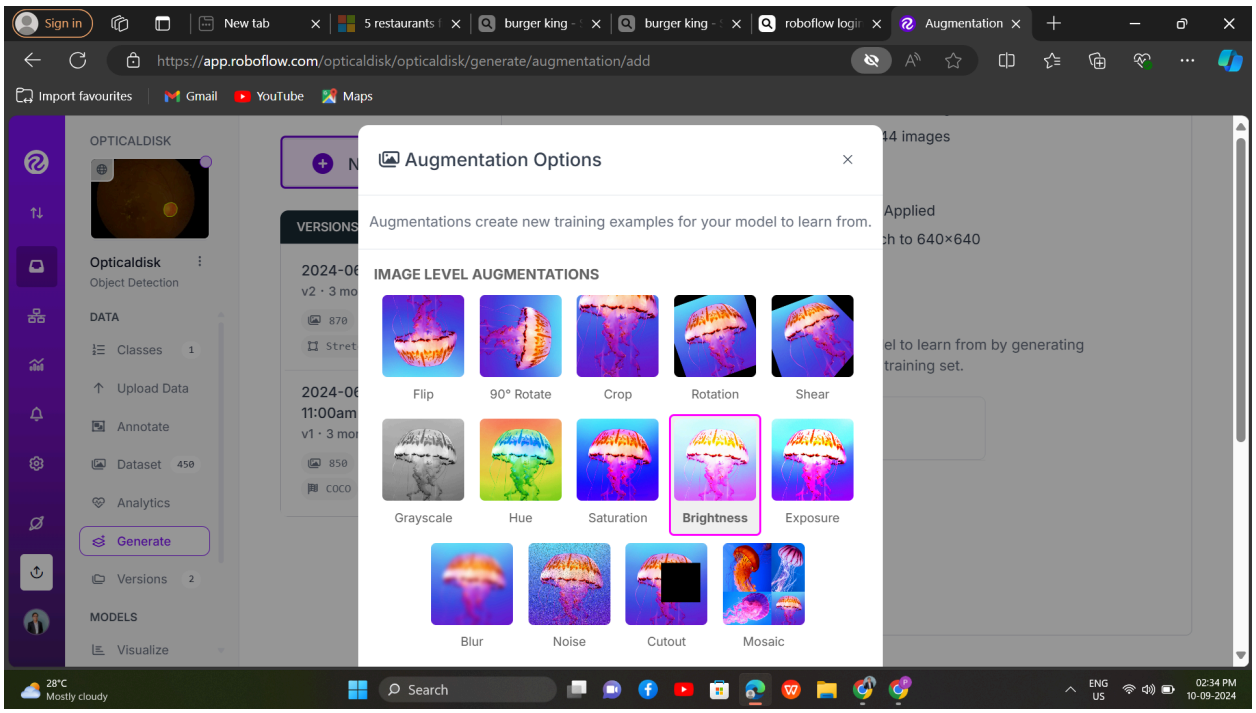
6. Review and Save Annotations
   - Once all your images are annotated, review your work.
   - You can re-adjust bounding boxes if necessary or add additional annotations.
   - After reviewing, save the annotations.



7. Preprocess Data

- Roboflow allows you to preprocess the images. You can resize, augment, or normalize your dataset to improve the performance of your model.
- Choose appropriate preprocessing settings based on your project needs.

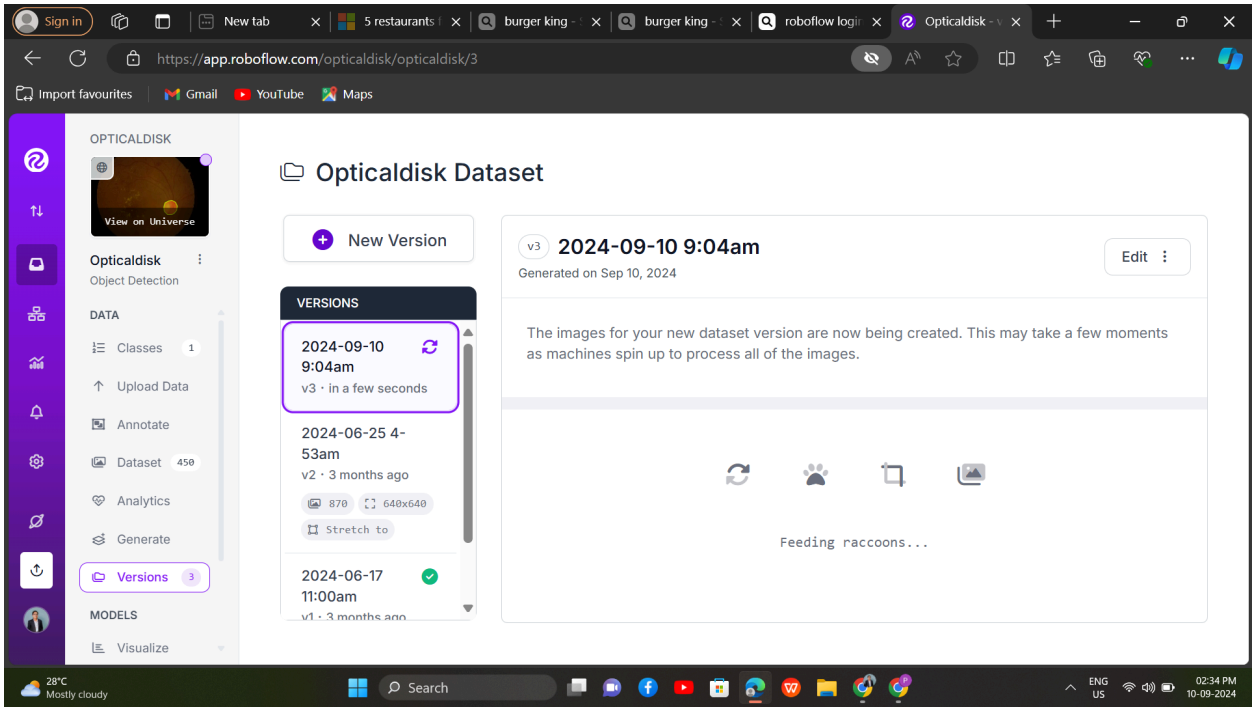


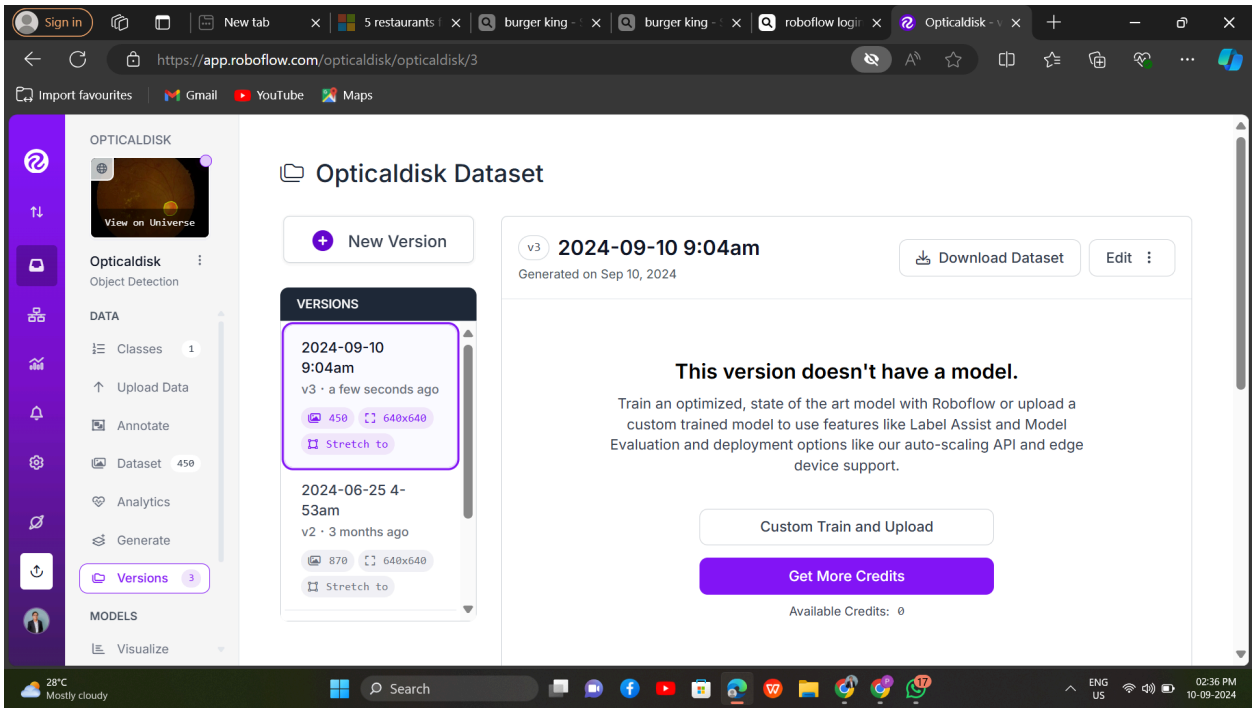


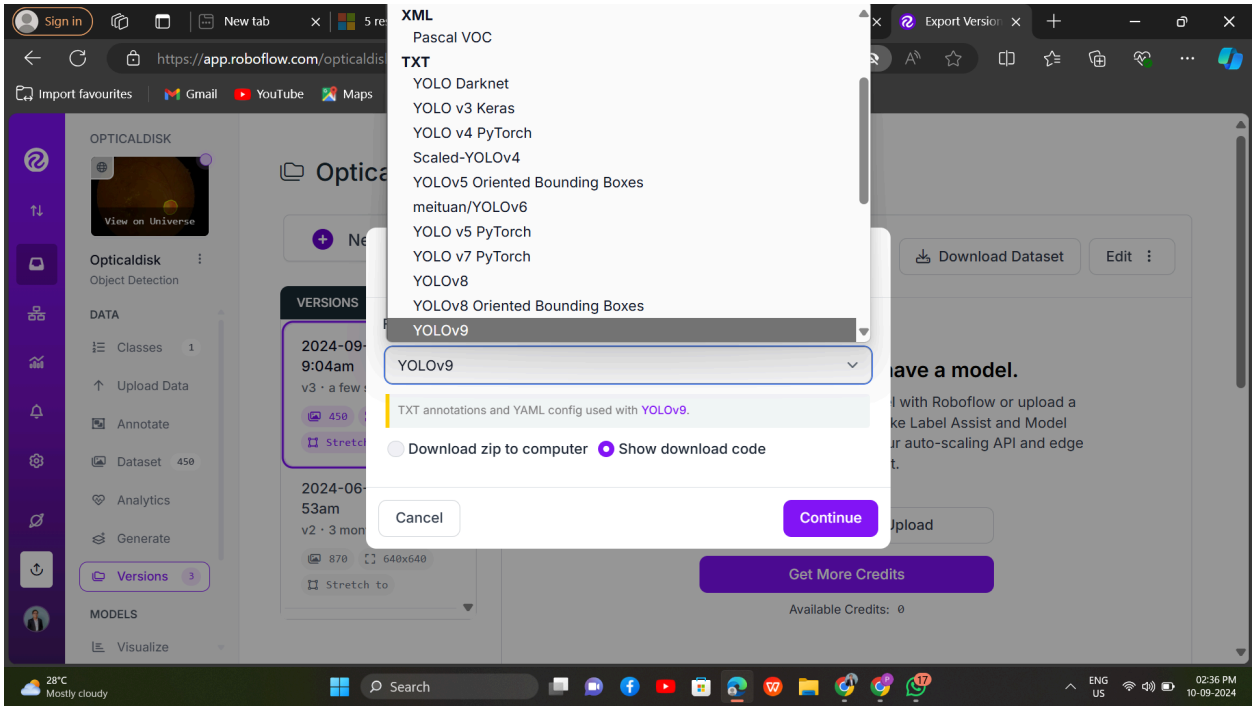
8. Export Dataset

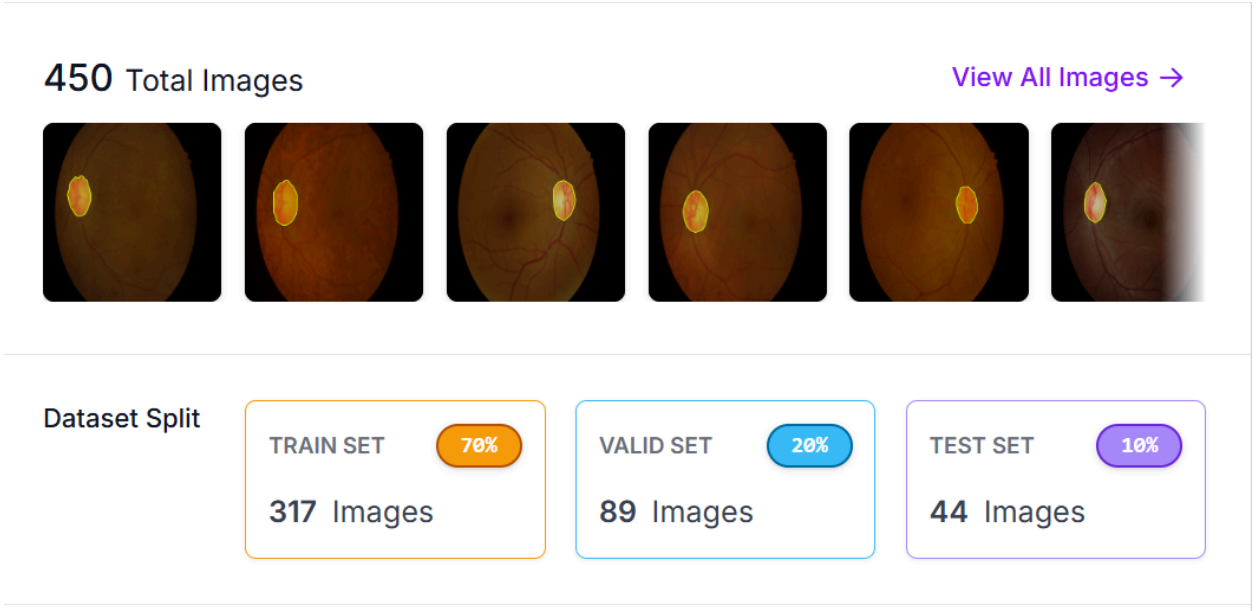
- After completing annotations and preprocessing, you can export the dataset.
- Choose your preferred format, such as TFRecord, COCO JSON, YOLOv5, or others.

- Download the dataset, which includes the images and the respective annotation files.

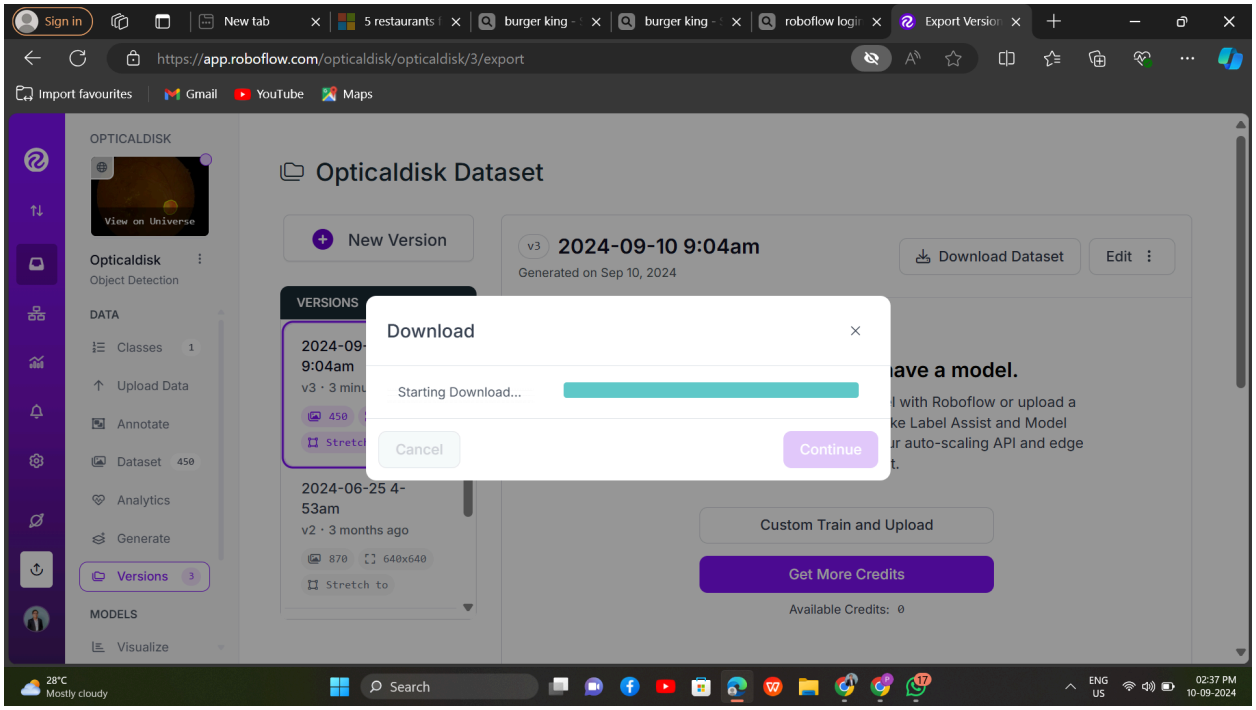








9. Download Labeled Data
   - Once your dataset is exported, download it to your local machine.
   - You will receive a zipped folder containing the images and annotation files in your chosen format.



Reference Link : https://roboflow.com/

Detron 2

Code :

```
# Install Roboflow and Detectron2
!pip install roboflow
!pip install pyyaml==5.1  # Specific version for compatibility
!pip install 'git+https://github.com/facebookresearch/detectron2.git'
```

Output:

```
Collecting roboflow
  Downloading roboflow-1.1.45-py3-none-any.whl.metadata (9.7 kB)
Requirement already satisfied: certifi in /usr/local/lib/python3.10/dist-packages (from roboflow) (2024.8.30)
Collecting idna==3.7 (from roboflow)
  Downloading idna-3.7-py3-none-any.whl.metadata (9.9 kB)
```

Line 1: Install Roboflow

bash
Copy code

```
!pip install roboflow
```

Explanation:
This command installs the `roboflow` Python package, which provides an interface to interact with the Roboflow platform. Roboflow is a tool that simplifies image dataset management, training, and deployment for computer vision models. By installing it, you can easily integrate with your Roboflow account, upload datasets, and download pre-processed versions ready for use in machine learning models.

Line 2: Install Specific Version of `pyyaml`

bash
Copy code

```
!pip install pyyaml==5.1
```

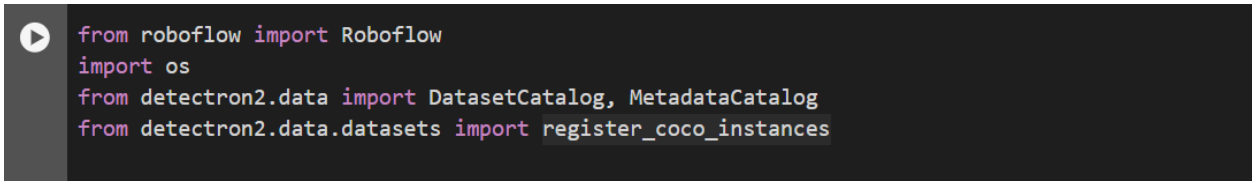
Explanation:
This command installs version 5.1 of the `pyyaml` package. `pyyaml` is a Python library for parsing and writing YAML (YAML Ain't Markup Language), which is often used for configuration files. The specific version `5.1` is installed because some libraries, like Detectron2, may have compatibility issues with newer versions of `pyyaml`, and installing an older version helps avoid errors during installation or execution.

Line 3: Install Detectron2 from GitHub

bash
Copy code
```
!pip install 'git+https://github.com/facebookresearch/detectron2.git'
```

Explanation:
This command installs the Detectron2 library directly from its GitHub repository using `pip`. Detectron2 is a powerful framework from Facebook AI Research (FAIR) for building high-performance object detection and segmentation models. By installing it through GitHub, you ensure that you're getting the latest version available in the repository. The `git+` prefix tells `pip` to install the package from a Git repository rather than the default Python Package Index (PyPI).

Code :



Line 1: Import Roboflow

python
Copy code
```
from roboflow import Roboflow
```

Explanation:
This line imports the `Roboflow` class from the `roboflow` package. Roboflow provides tools to work with image datasets and machine learning models, specifically for tasks like object detection, classification, and segmentation. By importing it, you can access its functionality to connect to the Roboflow platform, download datasets, and prepare them for use in training models.

Line 2: Import `os` Module

python
Copy code
```
import os
```

Explanation:
This line imports the built-in `os` module, which provides functions to interact with the operating system. The `os` module is useful for tasks such as handling file paths, managing environment variables, and performing operations like creating, reading, and writing to files. In the context of this code, it might be used to set environment variables or manage file paths for datasets.

Line 3: Import `DatasetCatalog` and `MetadataCatalog` from Detectron2

python
Copy code

```
from detectron2.data import DatasetCatalog, MetadataCatalog
```

Explanation:
This line imports two important classes from the `detectron2.data` module:

- `DatasetCatalog`: This class is used to register and retrieve datasets in Detectron2. It allows you to define custom datasets and register them for use in training and evaluation. Each dataset is associated with a unique name, which can then be used throughout the Detectron2 framework.
- `MetadataCatalog`: This class is used to store metadata about a dataset, such as class names, colors, and keypoints. Metadata is useful when visualizing predictions and for providing additional information about a dataset when it is used in a model.

Code:

```
# Initialize Roboflow
rf = Roboflow(api_key="aj1YNkFeRPD8FAf7DMpN")

# Access the specific project and version
project = rf.workspace("opticaldisk").project("opticaldisk")
version = project.version(2)

# Download the dataset in COCO format
dataset = version.download("coco")

# Print the dataset location
print(dataset.location)
```

Output:

```
loading Roboflow workspace...
loading Roboflow project...
Downloading Dataset Version Zip in Opticaldisk-2 to coco:: 100%|██████████| 21658/21658 [00:00<00:00, 26001.56it/s]

Extracting Dataset Version Zip to Opticaldisk-2 in coco:: 100%|██████████| 878/878 [00:00<00:00, 5553.20it/s]/content/Opticaldisk-2
```

Line 1: Initialize Roboflow

python
Copy code

```
rf = Roboflow(api_key="aj1YNkFeRPD8FAf7DMpN")
```

Explanation:
This line initializes an instance of the `Roboflow` class using your API key. The API key is a unique identifier that allows you to authenticate and access your projects and datasets on the Roboflow platform.

The initialized `rf` object is used to interact with Roboflow's services, such as downloading datasets or accessing specific projects.

Line 2: Access a Specific Workspace, Project, and Version

python
Copy code
```
project = rf.workspace("opticaldisk").project("opticaldisk")
```

Explanation:
Here, you are accessing a specific workspace and project within Roboflow. First, you call the `workspace("opticaldisk")` method to access a workspace named `"opticaldisk"`. Then, within that workspace, you access a project also named `"opticaldisk"`.

A workspace is a collection of projects, and each project typically corresponds to a specific computer vision task (e.g., object detection for optical disks). The `project` object now represents the project you want to work with.

Line 3: Select a Specific Version of the Project

python
Copy code
```
version = project.version(2)
```

Explanation:
This line specifies the version of the dataset you want to access. Each Roboflow project can have multiple versions, which might represent different iterations of a dataset (e.g., after additional labeling or corrections). Here, version `2` is being accessed. This ensures that the correct version of the dataset (which may have updated annotations or image data) is being used.

Line 4: Download the Dataset in COCO Format

python
Copy code
```
dataset = version.download("coco")
```

Explanation:
This line downloads the dataset in COCO format. The COCO format is a popular format used for object detection and segmentation tasks. It includes both image files and annotation files in JSON format, describing objects within the images (e.g., bounding boxes, segmentation masks). The dataset is stored in the `dataset` variable, which contains metadata and information about the downloaded files.

Roboflow allows you to export datasets in various formats (like YOLO, VOC, or COCO), and by specifying `"coco"`, you're instructing Roboflow to provide the dataset in the COCO format.

Line 5: Print Dataset Location

python
Copy code

```
print(dataset.location)
```

Explanation:
This line prints the location of the downloaded dataset on your local system. The `location` attribute of the `dataset` object holds the file path where the dataset was saved after downloading. By printing this, you can quickly find where the images and annotations are stored for further use in model training or evaluation.

Code:

```
# Paths to the dataset
train_json = os.path.join(dataset.location, "train/_annotations.coco.json")
train_images = os.path.join(dataset.location, "train")
val_json = os.path.join(dataset.location, "valid/_annotations.coco.json")
val_images = os.path.join(dataset.location, "valid")

# Register the dataset with Detectron2
register_coco_instances("opticaldisk_train", {}, train_json, train_images)
register_coco_instances("opticaldisk_val", {}, val_json, val_images)

# Verify the registration by printing the number of training images
dataset_dicts = DatasetCatalog.get("opticaldisk_train")
metadata = MetadataCatalog.get("opticaldisk_train")
print(f"Number of training images: {len(dataset_dicts)}")
```

Output:

```
WARNING:detectron2.data.datasets.coco:
Category ids in annotations are not in [1, #categories]! We'll apply a mapping for you.

Number of training images: 736
```

Line 1-4: Define Paths to the Dataset

python
Copy code

```
train_json = os.path.join(dataset.location,
"train/_annotations.coco.json")
train_images = os.path.join(dataset.location, "train")
```

```
val_json = os.path.join(dataset.location,
"valid/_annotations.coco.json")
val_images = os.path.join(dataset.location, "valid")
```

Explanation:

- `train_json`: This variable stores the path to the training annotations file in COCO format. The `os.path.join()` function constructs the path by combining the dataset location with the relative path to the annotations file.
- `train_images`: This variable holds the path to the directory containing the training images.
- `val_json`: Similar to `train_json`, this variable stores the path to the validation annotations file in COCO format.
- `val_images`: This variable holds the path to the directory containing the validation images.

These paths will be used later to register the dataset with Detectron2.

Line 6-8: Register the Dataset with Detectron2

python
Copy code

```
register_coco_instances("opticaldisk_train", {}, train_json,
train_images)
register_coco_instances("opticaldisk_val", {}, val_json, val_images)
```

Explanation:

- `register_coco_instances(name, metadata, json_file, image_root)`:
  - This function registers a dataset in COCO format with Detectron2, allowing it to be used for training and evaluation.
  - `"opticaldisk_train"`: This is the name given to the training dataset. It will be used to refer to this dataset later.
  - `{}`: An empty dictionary is passed for metadata, but you can include additional information like class names if desired.
  - `train_json`: The path to the training annotations file.
  - `train_images`: The path to the training images directory.
- The same registration process is applied for the validation dataset with the name `"opticaldisk_val"`.

These lines make the datasets available for use in Detectron2's training and evaluation processes.

Line 10-13: Verify the Registration

python
Copy code

```
dataset_dicts = DatasetCatalog.get("opticaldisk_train")
metadata = MetadataCatalog.get("opticaldisk_train")
print(f"Number of training images: {len(dataset_dicts)}")
```

Explanation:

- `DatasetCatalog.get("opticaldisk_train")`: This retrieves the registered training dataset as a list of dictionaries, where each dictionary contains details about an image (e.g., file name, annotations). The result is stored in the `dataset_dicts` variable.
- `MetadataCatalog.get("opticaldisk_train")`: This retrieves the metadata associated with the registered training dataset, stored in the `metadata` variable. This can include class names and other relevant information.
- `print(f"Number of training images: {len(dataset_dicts)}")`: This line prints the total number of training images registered in the dataset. The `len(dataset_dicts)` function returns the count of the dictionaries in `dataset_dicts`, representing the number of images.

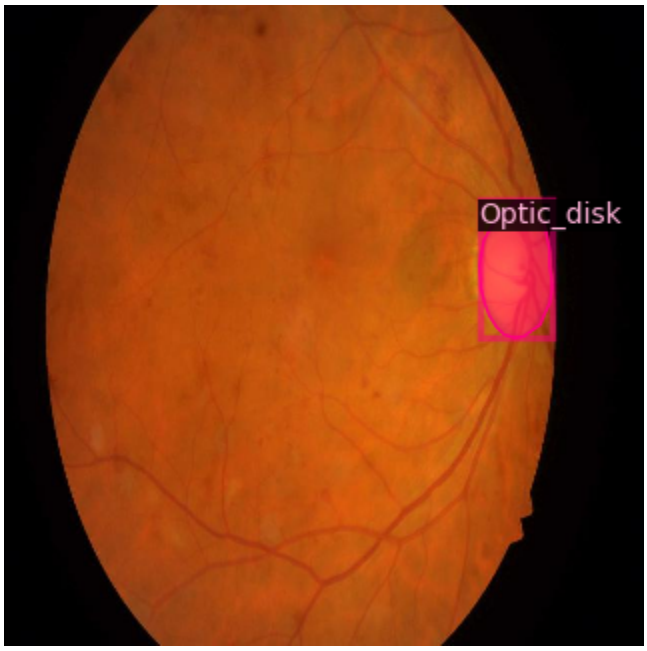
This verification step ensures that the datasets were registered correctly and allows you to confirm the number of training images available for the training process.

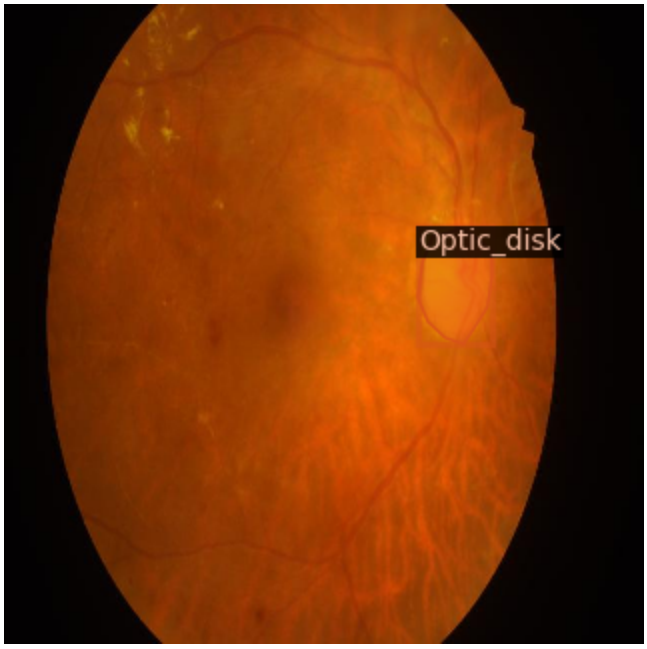
Code:

```
import random
import cv2
from detectron2.utils.visualizer import Visualizer
from google.colab.patches import cv2_imshow

# Visualize a few samples from the training set
for d in random.sample(dataset_dicts, 3):
    img = cv2.imread(d["file_name"])
    visualizer = Visualizer(img[:, :, ::-1], metadata=metadata, scale=0.5)
    vis = visualizer.draw_dataset_dict(d)
    cv2_imshow(vis.get_image()[:, :, ::-1])
```

Output:


Optic_disk


Optic_disk



Line 1-4: Import Libraries

python
Copy code

```
import random
import cv2
from detectron2.utils.visualizer import Visualizer
from google.colab.patches import cv2_imshow
```

Explanation:

- `random`: The `random` module is part of Python's standard library, and it's used here to select random samples from the dataset.
- `cv2`: This is the OpenCV library, commonly used for image processing. The `cv2.imread()` function reads an image from a file into memory.
- `Visualizer`: The `Visualizer` class from Detectron2 provides tools for visualizing datasets (such as bounding boxes, masks, and annotations) overlaid on images.
- `cv2_imshow`: This is a utility function specifically for Google Colab, allowing images to be displayed using OpenCV (`cv2.imshow()` doesn't work in Colab). It shows the image in the notebook.

Line 5: Loop Through Random Samples

python

Copy code
```
for d in random.sample(dataset_dicts, 3):
```

Explanation:
This line selects 3 random samples from the dataset (`dataset_dicts`) and iterates over them. The `random.sample()` function takes two arguments: the dataset and the number of samples to randomly select. Each sample `d` is a dictionary representing an image and its annotations from the training set.

Line 6: Load an Image Using OpenCV

python
Copy code
```
    img = cv2.imread(d["file_name"])
```

Explanation:
This line reads an image from the file system using its file path, which is stored in the `"file_name"` key of the dataset dictionary `d`. The image is loaded into memory as a NumPy array (`img`). OpenCV's `cv2.imread()` function reads the image in BGR format.

Line 7: Initialize the Visualizer

python
Copy code
```
    visualizer = Visualizer(img[:, :, ::-1], metadata=metadata,
scale=0.5)
```

Explanation:
This line initializes the Detectron2 `Visualizer` with the following parameters:

- `img[:, :, ::-1]`: The image array is converted from BGR (OpenCV's default format) to RGB (Detectron2's expected format) by reversing the channel order (`:: -1`).
- `metadata`: The dataset's metadata (class labels, colors, etc.) is passed to the visualizer for proper annotation overlay.
- `scale=0.5`: The scale parameter adjusts the size of the visualized image by 50%, reducing its dimensions for easier display.

Line 8: Draw the Annotations

python
Copy code
```
    vis = visualizer.draw_dataset_dict(d)
```

Explanation:
This line uses the `visualizer` object to draw the dataset annotations (such as bounding boxes, segmentation masks, and labels) over the image. The annotations are stored in the dataset dictionary `d`. The result is stored in `vis`, which is an image object with annotations drawn over it.

Line 9: Display the Image with Annotations

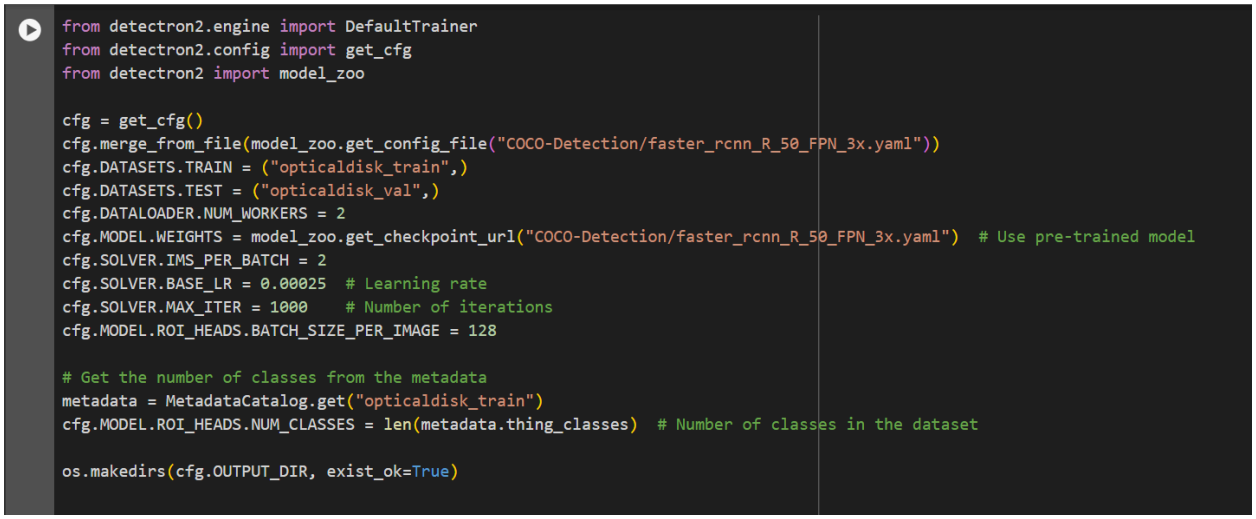
python
Copy code

```
cv2_imshow(vis.get_image()[:, :, ::-1])
```

Explanation:
This line displays the annotated image in Google Colab. The following steps happen:

- `vis.get_image()`: Retrieves the annotated image (with bounding boxes, labels, etc.).
- `[:, :, ::-1]`: Converts the image from RGB back to BGR for displaying with OpenCV.
- `cv2_imshow()`: Displays the image in the notebook.

Code:



```
from detectron2.engine import DefaultTrainer
from detectron2.config import get_cfg
from detectron2 import model_zoo

cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/faster_rcnn_R_50_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("opticaldisk_train",)
cfg.DATASETS.TEST = ("opticaldisk_val",)
cfg.DATALOADER.NUM_WORKERS = 2
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Detection/faster_rcnn_R_50_FPN_3x.yaml")  # Use pre-trained model
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.SOLVER.BASE_LR = 0.00025  # Learning rate
cfg.SOLVER.MAX_ITER = 1000    # Number of iterations
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 128

# Get the number of classes from the metadata
metadata = MetadataCatalog.get("opticaldisk_train")
cfg.MODEL.ROI_HEADS.NUM_CLASSES = len(metadata.thing_classes)  # Number of classes in the dataset

os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
```

Line 1-3: Importing Required Libraries

python
Copy code

```
from detectron2.engine import DefaultTrainer
from detectron2.config import get_cfg
```

```
from detectron2 import model_zoo
```

Explanation:

- `DefaultTrainer`: A default training class provided by Detectron2, which handles the training loop, model loading, evaluation, and more.
- `get_cfg`: A function to create a default configuration object, which will store all the hyperparameters and settings for training the model.
- `model_zoo`: Provides pre-built configurations and weights for various models. You can use this to quickly load popular architectures like Faster R-CNN, Mask R-CNN, etc.

Line 4: Initialize the Configuration Object

python
Copy code

```
cfg = get_cfg()
```

Explanation:

This line initializes the configuration object (`cfg`), which stores all the settings related to the model, data, and training process. You will modify this configuration to suit your dataset and model settings.

Line 5: Merge Predefined Model Configuration from Model Zoo

python
Copy code

```
cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/faster_rcnn_R_50_FPN_3x.yaml"))
```

Explanation:

This line merges the configuration of a pre-trained Faster R-CNN model from the Detectron2 model zoo. The specific model here is Faster R-CNN with a ResNet-50 backbone and Feature Pyramid Network (FPN), trained for 3x schedule on the COCO dataset. By merging this configuration, you import all the default settings for this model architecture.

Line 6-7: Specify Training and Validation Datasets

python
Copy code

```
cfg.DATASETS.TRAIN = ("opticaldisk_train",)
cfg.DATASETS.TEST = ("opticaldisk_val",)
```

Explanation:

- `cfg.DATASETS.TRAIN`: This sets the training dataset to `"opticaldisk_train"`, which you registered earlier.
- `cfg.DATASETS.TEST`: This sets the validation dataset to `"opticaldisk_val"`. Even though it's called "TEST," it will be used during the evaluation phase after each training iteration.

These datasets are specified by the names given when you registered them with `register_coco_instances()`.

Line 8: Set Number of Data Loader Workers

python
Copy code

```
cfg.DATALOADER.NUM_WORKERS = 2
```

Explanation:

This sets the number of CPU workers to use for loading data. Here, you are specifying 2 workers, which will load data in parallel during training. Increasing this value can speed up training, depending on your machine's CPU capabilities.

Line 9: Load Pre-Trained Weights

python
Copy code

```
cfg.MODEL.WEIGHTS =
model_zoo.get_checkpoint_url("COCO-Detection/faster_rcnn_R_50_FPN_3x.y
aml")
```

Explanation:

This line loads the pre-trained weights for the model specified in the previous configuration (`faster_rcnn_R_50_FPN_3x.yaml`). Pre-trained weights allow you to start from a model that has already been trained on the COCO dataset, which can speed up training and improve accuracy by transferring knowledge to your custom dataset.

Line 10-12: Set Training Hyperparameters

python
Copy code

```
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.SOLVER.BASE_LR = 0.00025
```

```
cfg.SOLVER.MAX_ITER = 1000
```

Explanation:

- `cfg.SOLVER.IMS_PER_BATCH = 2`: This sets the number of images per batch during training. A lower batch size like 2 is often necessary for limited GPU memory.
- `cfg.SOLVER.BASE_LR = 0.00025`: The base learning rate for training. This determines the step size for updating the model weights during training. Smaller learning rates lead to more stable training, while larger values may speed up convergence but increase the risk of instability.
- `cfg.SOLVER.MAX_ITER = 1000`: Specifies the maximum number of training iterations. After 1000 iterations, the training will stop. You can adjust this based on your dataset size and training needs.

Line 13: Set Batch Size per Image for ROI Heads

python
Copy code

```
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 128
```

Explanation:

This sets the batch size for ROI heads. The ROI heads process the regions of interest (ROI) generated by the Region Proposal Network (RPN) during training. A value of 128 indicates that 128 regions per image will be used during each training step. This value directly affects memory usage and speed.

Line 14-15: Set the Number of Classes in the Dataset

python
Copy code

```
metadata = MetadataCatalog.get("opticaldisk_train")
cfg.MODEL.ROI_HEADS.NUM_CLASSES = len(metadata.thing_classes)
```

Explanation:

- `MetadataCatalog.get("opticaldisk_train")`: Retrieves the metadata for the training dataset (`opticaldisk_train`). This metadata contains information such as class labels.
- `cfg.MODEL.ROI_HEADS.NUM_CLASSES`: This sets the number of classes for the model's ROI heads based on the number of object categories in the dataset. Here, `len(metadata.thing_classes)` retrieves the number of unique object classes in the training dataset.

Line 16: Create the Output Directory

python
Copy code
```
os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
```

Explanation:

This line creates the directory where the model's checkpoints and logs will be saved during training. If the directory already exists, the `exist_ok=True` flag prevents errors. By default, the output will be saved to `cfg.OUTPUT_DIR`.

Code:

```
trainer = DefaultTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```

Output:

```
[09/20 06:25:53 d2.data.datasets.coco]: Loaded 90 images in COCO format from /content/Opticaldisk-2/valid/_annotations.coco.json
[09/20 06:25:53 d2.data.build]: Distribution of instances among all 2 categories:
|  category  | #instances   |  category  | #instances   |
|:----------:|:-------------|:----------:|:-------------|
|  Diabetic  | 0            | Optic_disk | 90           |
|            |              |            |              |
|   total    | 90           |            |              |
[09/20 06:25:53 d2.data.dataset_mapper]: [DatasetMapper] Augmentations used in inference: [ResizeShortestEdge(short_edge_length=(800, 800), max_size=1333, sam
[09/20 06:25:53 d2.data.common]: Serializing the dataset using: <class 'detectron2.data.common._TorchSerializedList'>
[09/20 06:25:53 d2.data.common]: Serializing 90 elements to byte tensors and concatenating them all ...
[09/20 06:25:53 d2.data.common]: Serialized dataset takes 0.05 MiB
```

Line 1: Initialize the Trainer

python
Copy code
```
trainer = DefaultTrainer(cfg)
```

Explanation:

- `DefaultTrainer(cfg)`: This initializes the `DefaultTrainer` class from Detectron2 using the configuration (`cfg`) you defined earlier. The `DefaultTrainer` is a high-level API that handles the entire training loop, including loading data, managing model checkpoints, and running evaluation.
- `cfg`: This configuration object contains all the settings needed for training the model, such as the dataset, model architecture, hyperparameters, and output directory.

By creating an instance of `DefaultTrainer`, you are setting up the training pipeline with the specified configuration.

Line 2: Load Checkpoints (If Any)

python
Copy code
`trainer.resume_or_load(resume=False)`

Explanation:

- `resume_or_load(resume=False)`: This method controls how the training starts:
  - If `resume=False`, training will start from scratch, loading pre-trained weights (if specified) but ignoring any existing checkpoints.
  - If `resume=True`, it will try to resume training from an existing checkpoint saved in the output directory. This is useful if training was interrupted and you want to continue from where it left off.

Here, `resume=False` means the training will start fresh using the initial pre-trained weights, not from a previously saved checkpoint.

Line 3: Start the Training Process

python
Copy code
`trainer.train()`

Explanation:

- `train()`: This method starts the training loop. It handles all aspects of the training, including loading the data, feeding it into the model, performing forward and backward passes, and updating the model's weights. It also handles model checkpoints and, if configured, runs evaluation after each iteration or epoch.

During the training loop, it:

- Reads the data (images and annotations) from the dataset you registered (`opticaldisk_train`).
- Uses the Faster R-CNN architecture to detect objects in the images.
- Updates the model's parameters based on the losses calculated during the forward pass.
- Saves checkpoints and logs progress, such as training loss, to the specified output directory.

Code:

```
from detectron2.evaluation import COCOEvaluator, inference_on_dataset
from detectron2.data import build_detection_test_loader

evaluator = COCOEvaluator("opticaldisk_val", cfg, False, output_dir="./output/")
val_loader = build_detection_test_loader(cfg, "opticaldisk_val")
inference_on_dataset(trainer.model, val_loader, evaluator)
```

Output:

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.805
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.830
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.830
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.830
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.823
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.836
[09/20 06:26:01 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs  |  APm   |  APl   |
|:------:|:------:|:------:|:-----:|:------:|:------:|
| 77.996 | 97.764 | 97.764 |  nan  | 77.318 | 80.545 |
[09/20 06:26:01 d2.evaluation.coco_evaluation]: Some metrics cannot be computed and is shown as NaN.
[09/20 06:26:01 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category   | AP   | category    | AP     |
|:-----------|:-----|:------------|:-------|
| Diabetic   | nan  | Optic_disk | 77.996 |
OrderedDict([('bbox',
              {'AP': 77.99566890292364,
               'AP50': 97.76442269489327,
               'AP75': 97.76442269489327,
               'APs': nan,
               'APm': 77.3181678698544,
               'APl': 80.54483576420533,
               'AP-Diabetic': nan,
               'AP-Optic_disk': 77.99566890292364})])
```

Line 1-2: Importing Required Functions

python
Copy code

```
from detectron2.evaluation import COCOEvaluator, inference_on_dataset
from detectron2.data import build_detection_test_loader
```

Explanation:

- `COCOEvaluator`: This class is used to evaluate the model's performance on a dataset using COCO metrics, such as Average Precision (AP) and Intersection over Union (IoU) for object detection. COCO is a popular dataset format for object detection tasks, and these metrics are widely used for performance evaluation.
- `inference_on_dataset`: This function runs the trained model on a given dataset, computes predictions, and evaluates the model's performance using the evaluator you define.

- `build_detection_test_loader`: This function creates a data loader for the test dataset. It's responsible for fetching and batching the validation data during the evaluation process.

Line 3: Initialize the Evaluator

python
Copy code
```
evaluator = COCOEvaluator("opticaldisk_val", cfg, False,
output_dir="./output/")
```

Explanation:

- `COCOEvaluator("opticaldisk_val", cfg, False, output_dir="./output/")`:
  - `"opticaldisk_val"`: This is the name of the validation dataset. It tells the evaluator which dataset to use for evaluation, in this case, the validation dataset you registered (`opticaldisk_val`).
  - `cfg`: The configuration object, which includes important settings like the dataset, model, and output directory.
  - `False`: This indicates that you are not using distributed evaluation (i.e., evaluating on multiple GPUs). If you were running this on multiple GPUs, you would set this to `True`.
  - `output_dir="./output/"`: The directory where evaluation results and metrics will be saved. This folder will store the evaluation output, such as logs, metrics, and visualizations (if applicable).

Line 4: Build the Validation Data Loader

python
Copy code
```
val_loader = build_detection_test_loader(cfg, "opticaldisk_val")
```

Explanation:

- `build_detection_test_loader(cfg, "opticaldisk_val")`: This function builds a data loader for the validation dataset (`opticaldisk_val`). The loader will handle loading images and annotations from the validation set in batches, which will be passed through the model for inference.
  - `cfg`: The configuration object, which contains dataset-related settings like batch size, number of workers, and augmentation (if any).
  - `"opticaldisk_val"`: The registered name of the validation dataset, which you defined earlier.

This loader allows the model to process the validation dataset in a structured way for evaluation.

Line 5: Perform Inference and Evaluate the Model

python
Copy code

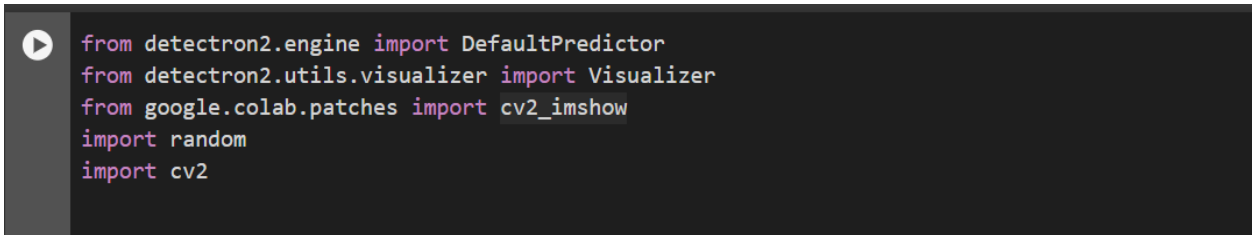
```
inference_on_dataset(trainer.model, val_loader, evaluator)
```

Explanation:

- `inference_on_dataset(trainer.model, val_loader, evaluator)`:
  - `trainer.model`: This is the trained model, which will be used to make predictions on the validation data.
  - `val_loader`: The data loader created earlier, which feeds the validation data to the model for inference.
  - `evaluator`: The COCOEvaluator object, which will compute evaluation metrics like precision, recall, and mAP (mean Average Precision).

This function runs the model on the validation dataset, generates predictions, and then passes those predictions to the evaluator. The evaluator compares the model's predictions with the ground truth annotations to compute performance metrics, which are logged and saved in the output directory.

Code:



```
from detectron2.engine import DefaultPredictor
from detectron2.utils.visualizer import Visualizer
from google.colab.patches import cv2_imshow
import random
import cv2
```

Line 1: Import DefaultPredictor

python

Copy code

```
from detectron2.engine import DefaultPredictor
```

Explanation:

- `DefaultPredictor`: This is a convenient wrapper provided by Detectron2 to make predictions with a trained model. It simplifies the inference process by managing the model's configuration, loading weights, and handling preprocessing/postprocessing of images.
- After initializing the `DefaultPredictor` with a configuration object (`cfg`), you can use it to run inference on images, making predictions about object locations, classes, and other outputs from the model.

Line 2: Import Visualizer

python

Copy code

```
from detectron2.utils.visualizer import Visualizer
```

Explanation:

- `Visualizer`: This is a utility provided by Detectron2 that helps visualize the predictions (e.g., bounding boxes, segmentation masks) made by the model. It draws bounding boxes, segmentation masks, labels, and keypoints on images, making it easy to understand and interpret the model's predictions.
- This tool is commonly used for displaying object detection results in images.

Line 3: Import Display Function for Colab

python

Copy code

```
from google.colab.patches import cv2_imshow
```

Explanation:

- `cv2_imshow`: This is a function from Google Colab's utility library that is specifically designed to display images using OpenCV within Colab notebooks.
- OpenCV's default `cv2.imshow()` function doesn't work in Google Colab. Instead, you use `cv2_imshow` to display images after processing them (such as displaying images with bounding boxes or masks drawn on them by the model).

Line 4: Import Random Module

python

Copy code

```
import random
```

Explanation:

- `random`: This is Python's built-in module for generating random numbers and performing random operations, such as selecting random elements from a list.
- In the context of this code, `random` will likely be used to randomly select a few images from the dataset to visualize the model's predictions.

Line 5: Import OpenCV

python

Copy code

```
import cv2
```

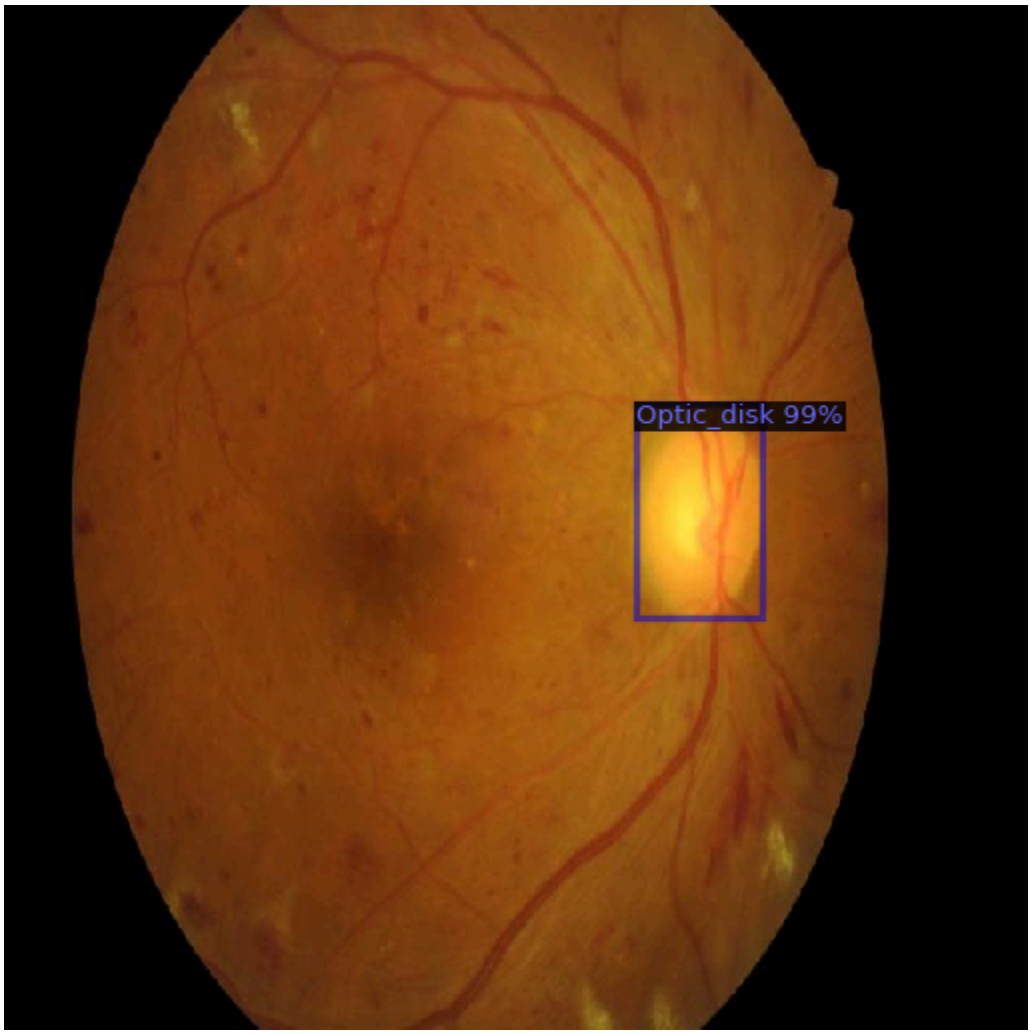
Explanation:

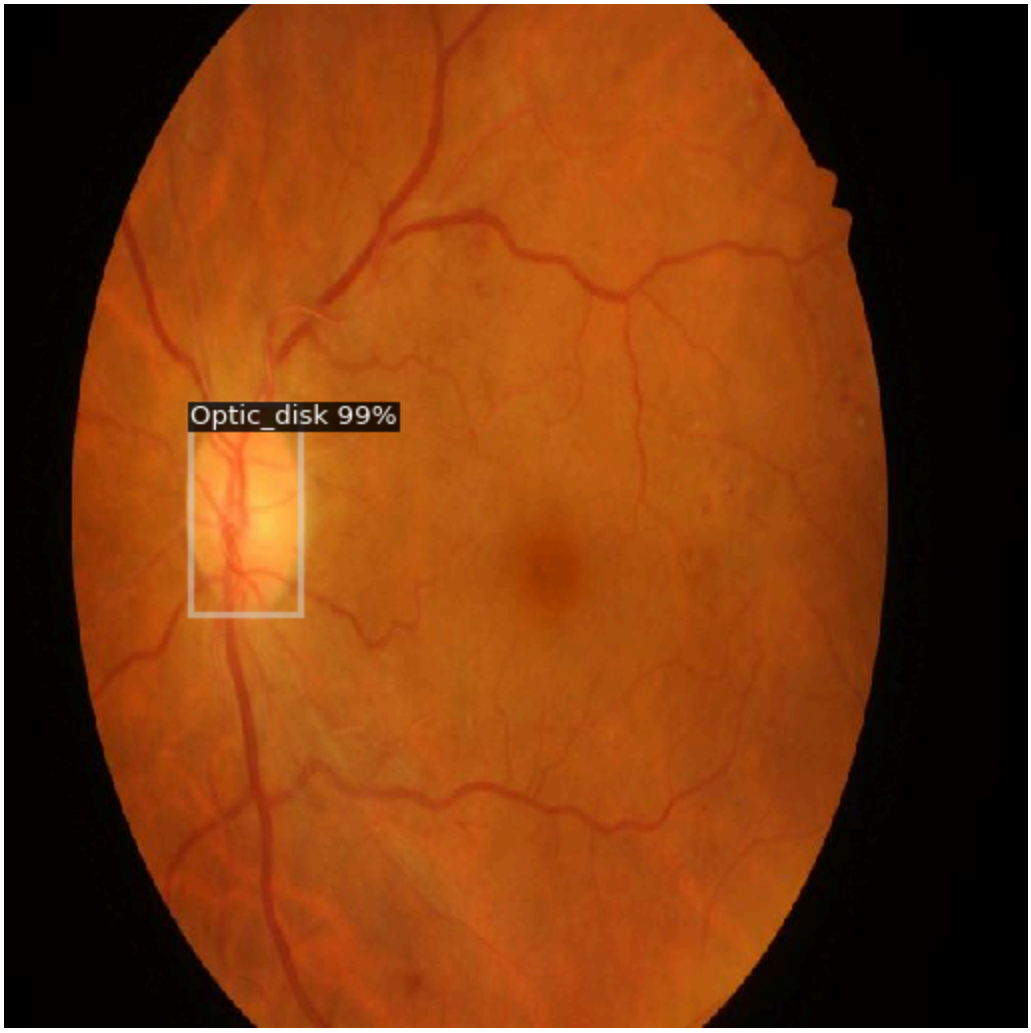
- `cv2`: This is the OpenCV library, which is used for image processing tasks. OpenCV provides functions to read, display, and manipulate images.
- In this code, `cv2` will be used to read images from disk (`cv2.imread()`), perform image transformations, and display or save the output with predictions.

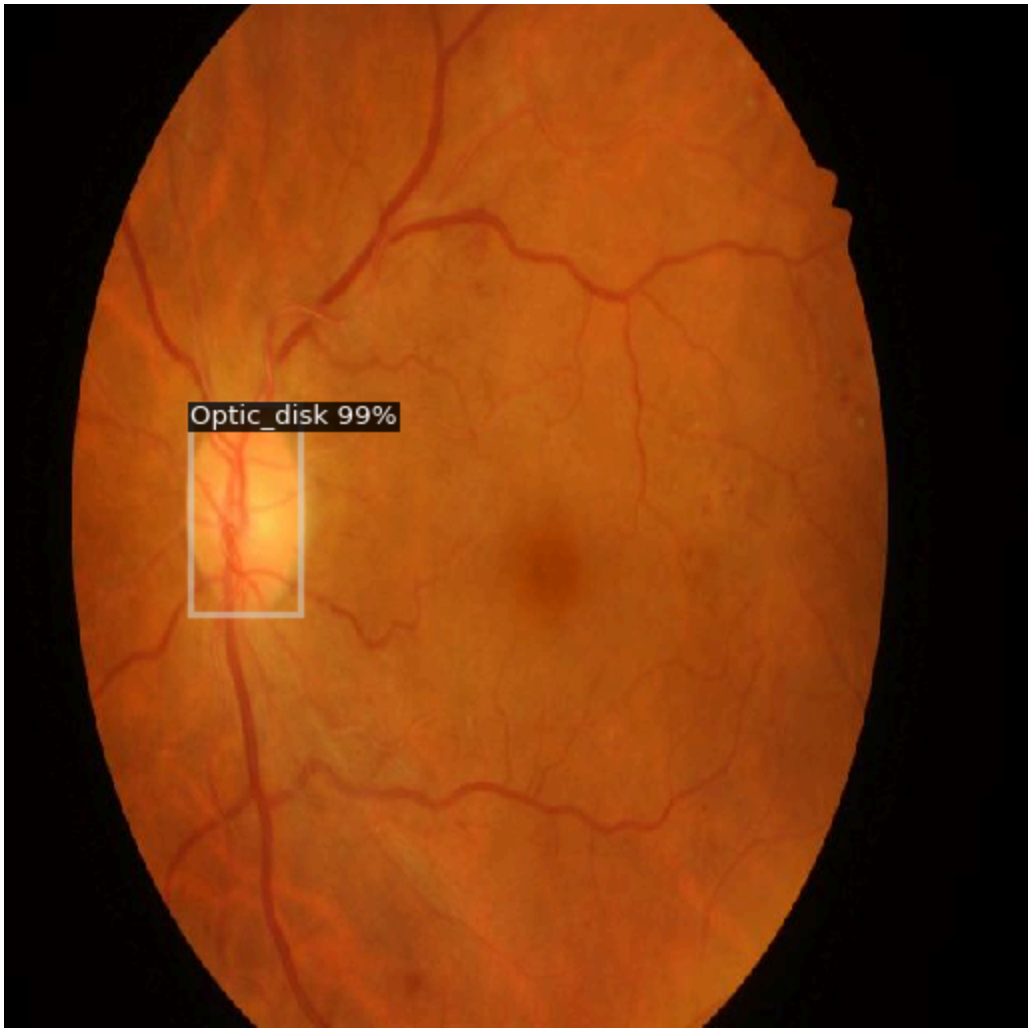
Code:

```
# Set the model weights to the final trained model
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5   # Set a threshold for this model
predictor = DefaultPredictor(cfg)

# Visualize some predictions
dataset_dicts = DatasetCatalog.get("opticaldisk_val")
for d in random.sample(dataset_dicts, 3):
    im = cv2.imread(d["file_name"])
    outputs = predictor(im)

    # Visualize the predictions
    v = Visualizer(im[:, :, ::-1], metadata=metadata, scale=0.8)
    out = v.draw_instance_predictions(outputs["instances"].to("cpu"))
    cv2_imshow(out.get_image()[:, :, ::-1])
```

Output:




Optic_disk 99%



Line 1: Set the Model Weights

python
Copy code

```
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")
```

Explanation:

- `cfg.MODEL.WEIGHTS`: This specifies the path to the weights (model parameters) for the trained model.
- `os.path.join(cfg.OUTPUT_DIR, "model_final.pth")`: This constructs the path to the final model weights file (`model_final.pth`) that was saved after training. It assumes the model weights are located in the `OUTPUT_DIR` specified in the configuration object (`cfg`).

This line sets the model to use the final trained weights, allowing it to make predictions based on the learned parameters.

Line 2: Set a Confidence Threshold for Predictions

python
Copy code
```
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5
```

Explanation:

- `SCORE_THRESH_TEST`: This is the confidence threshold for predictions. Only predictions with a score higher than this threshold (in this case, 0.5 or 50%) will be displayed.
- By setting the threshold to 0.5, the model will filter out less confident predictions and only return predictions where the model is at least 50% confident about the object it has detected.

Line 3: Initialize the Predictor

python
Copy code
```
predictor = DefaultPredictor(cfg)
```

Explanation:

- `DefaultPredictor(cfg)`: This initializes the `DefaultPredictor` class, which is a simplified way to run inference on a model. It uses the model configuration (`cfg`) to load the trained model and handles all preprocessing and postprocessing for predictions.
- The predictor object is now ready to take in an image and return predictions like bounding boxes, class labels, and confidence scores.

Here's a detailed explanation of the code that loads a trained model and visualizes its predictions on random samples from the validation dataset:

Line 4: Get the Validation Dataset

python
Copy code
```
dataset_dicts = DatasetCatalog.get("opticaldisk_val")
```

Explanation:

- `DatasetCatalog.get("opticaldisk_val")`: This retrieves the validation dataset (`opticaldisk_val`) that was registered earlier. The dataset is returned in a list of dictionaries, where each dictionary contains details about an image (e.g., file name, annotations, etc.).
- This line fetches the data required for validation, allowing you to run the model's predictions on the validation images.

Line 5-6: Loop Over Random Validation Samples

python
Copy code

```
for d in random.sample(dataset_dicts, 3):
    im = cv2.imread(d["file_name"])
```

Explanation:

- `random.sample(dataset_dicts, 3)`: This selects 3 random samples from the validation dataset to visualize the predictions. You can adjust this number if you want more or fewer samples.
- `cv2.imread(d["file_name"])`: Reads the image file from the dataset using OpenCV. The file path to each image is stored in the `file_name` key of each dictionary in `dataset_dicts`.

This loop processes 3 random images from the validation dataset for visualization.

Line 7: Make Predictions on the Image

python
Copy code

```
outputs = predictor(im)
```

Explanation:

- `predictor(im)`: This runs the trained model on the image (`im`) and returns the predictions.
- `outputs`: This contains the model's predictions for the input image, including bounding boxes, class labels, and confidence scores.

The model's predictions are stored in the `outputs` object.

Line 8-9: Initialize the Visualizer and Draw Predictions

python
Copy code

```
v = Visualizer(im[:, :, ::-1], metadata=metadata, scale=0.8)
```

```
out = v.draw_instance_predictions(outputs["instances"].to("cpu"))
```

Explanation:

- `Visualizer(im[:, :, ::-1], metadata=metadata, scale=0.8)`:
  - `im[:, :, ::-1]`: This reorders the color channels from BGR (OpenCV's format) to RGB (expected by the `Visualizer`).
  - `metadata=metadata`: The metadata contains information about the dataset, such as class names, which are used to label the predictions.
  - `scale=0.8`: This scales the image slightly for display. You can adjust the scale to make the output smaller or larger.
- `v.draw_instance_predictions(outputs["instances"].to("cpu"))`: This draws the model's predictions (bounding boxes, class labels, etc.) on the image.
  - `outputs["instances"]`: This contains the predicted objects (bounding boxes, masks, etc.).
  - `.to("cpu")`: Moves the predictions to the CPU for processing (since inference might have been done on a GPU).

Line 10: Display the Image with Predictions

python
Copy code

```
cv2_imshow(out.get_image()[:, :, ::-1])
```

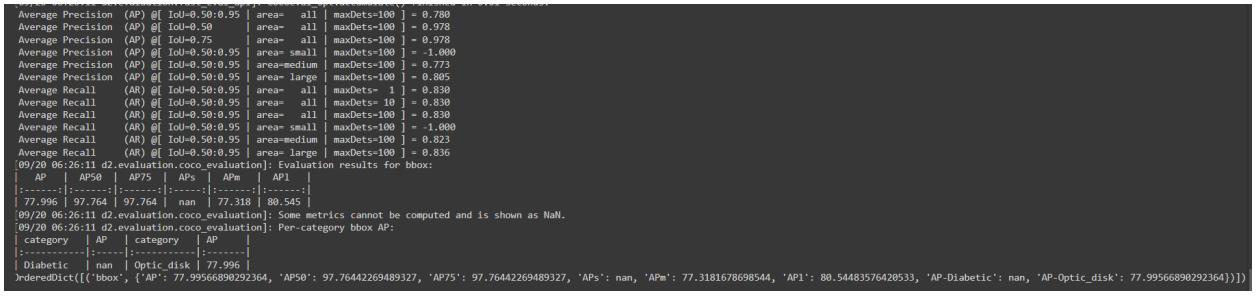
Explanation:

- `out.get_image()[:, :, ::-1]`: This retrieves the image with drawn predictions from the `Visualizer` and converts it back to BGR format for OpenCV.
- `cv2_imshow()`: Displays the image in Google Colab. Since `cv2.imshow()` doesn't work in Colab, you use this special function to display the image in the notebook.

Code:

```
from detectron2.evaluation import COCOEvaluator, inference_on_dataset
from detectron2.data import build_detection_test_loader

# Evaluate the model on the validation set
evaluator = COCOEvaluator("opticaldisk_val", cfg, False, output_dir="./output/")
val_loader = build_detection_test_loader(cfg, "opticaldisk_val")
metrics = inference_on_dataset(trainer.model, val_loader, evaluator)

# Print metrics to see the results
print(metrics)
```

Output:



Line 1-2: Import Required Functions

python
Copy code

```
from detectron2.evaluation import COCOEvaluator, inference_on_dataset
from detectron2.data import build_detection_test_loader
```

Explanation:

- `COCOEvaluator`: This class is used to evaluate the model's performance on a dataset using COCO metrics, such as Average Precision (AP) and Intersection over Union (IoU) for object detection.
- `inference_on_dataset`: This function runs inference on a dataset using a specified model and evaluator, calculating various performance metrics based on the predictions.

Line 4: Initialize the Evaluator

python
Copy code

```
evaluator = COCOEvaluator("opticaldisk_val", cfg, False,
output_dir="./output/")
```

Explanation:

- `COCOEvaluator("opticaldisk_val", cfg, False, output_dir="./output/")`:
  - `"opticaldisk_val"`: This is the name of the validation dataset. It tells the evaluator which dataset to use for evaluation.
  - `cfg`: The configuration object that contains settings for the evaluation.
  - `False`: Indicates that distributed evaluation is not being used (i.e., not evaluating on multiple GPUs).
  - `output_dir="./output/"`: The directory where evaluation results will be saved.

This line initializes the evaluator that will be used to compute the model's performance metrics.

Line 5: Build the Validation Data Loader

python
Copy code
```
val_loader = build_detection_test_loader(cfg, "opticaldisk_val")
```

Explanation:

- `build_detection_test_loader(cfg, "opticaldisk_val")`: This function creates a data loader for the validation dataset. It prepares batches of images and annotations, making it easy for the model to process them during evaluation.
    - `cfg`: The configuration object containing settings for the data loader.
    - `"opticaldisk_val"`: The name of the validation dataset.

This line sets up the data loader, which is essential for feeding data into the model during evaluation.

Line 6: Run Inference and Evaluation

python
Copy code
```
metrics = inference_on_dataset(trainer.model, val_loader, evaluator)
```

Explanation:

- `inference_on_dataset(trainer.model, val_loader, evaluator)`:
    - `trainer.model`: The trained model that will be used to make predictions on the validation dataset.
    - `val_loader`: The data loader created for the validation dataset, which fetches batches of images.
    - `evaluator`: The `COCOEvaluator` object, which will compute various evaluation metrics based on the model's predictions.

This function runs the model on the validation dataset, generates predictions, and evaluates these predictions using the COCO metrics defined in the evaluator. The results are stored in the `metrics` variable.

Line 8: Print the Metrics

python
Copy code
```
print(metrics)
```

Explanation:

- `print(metrics)`: This outputs the evaluation metrics to the console. The `metrics` variable contains information such as:
    - Average Precision (AP) at different IoU thresholds (e.g., AP, AP@0.5, AP@0.75).
    - Average Recall (AR).
    - Other statistics relevant to the performance of the model on the validation dataset.

By printing the metrics, you can assess how well the model performed on the validation set, gaining insights into its strengths and weaknesses in object detection tasks.

List of issues

List of issues faced during the development process how they are tackled whether they are resolved or not .

1.Duplicate Installation Commands:

- Problem: Multiple installation commands for roboflow were present.
- Resolution: Removed the redundant installation commands to streamline the setup process.

2.Dataset Download Path:

- Problem: Incorrect dataset paths or unexpected dataset structure might cause issues.
- Resolution: Added checks to print and verify the dataset location and file paths. Ensured that paths matched the dataset's structure by inspecting the directory contents.

3.Dataset Registration:

- Problem: Potential conflicts if the dataset was already registered.
- Resolution: Checked if the dataset was already registered and removed it if necessary before re-registering with unique names.

4.Annotation File Paths:

- Problem: The script might fail if the annotation files were not found at the expected paths.
- Resolution: Added checks to verify the existence of annotation files and raised informative errors if they were missing.

5.Visualization of Green Channel Images:

- Problem: Green channel images might not display correctly with annotations due to their single-channel nature.
- Resolution: Used a grayscale colormap (cmap='gray') for displaying green channel images to handle the single-channel nature appropriately.

6.Mouse Event Handling for Drawing Circles:

- Problem: Mouse events for drawing circles on images might not function as expected in different environments.
- Resolution: Provided a mechanism to load images from either local paths or URLs. Included resizing of the image to fit screen dimensions for better visualization.

7.Matplotlib and OpenCV Integration:

- Problem: Combining Matplotlib with OpenCV for interactive features might have compatibility issues.
- Resolution: Used Matplotlib to display images and annotations in a way compatible with OpenCV's image manipulation.

8.File Path and Directory Creation:

- Problem: Folder creation for saving images might fail due to permission issues or incorrect paths.
- Resolution: Added checks to create directories if they did not exist and printed confirmation messages.

9.Library Version Conflicts:

- Problem: Potential conflicts between different library versions (e.g., detectron2, torch, torchvision).
- Resolution: Ensured installation of compatible versions by specifying the necessary library versions or using the latest versions.

10.Error Handling and Debugging:

- Problem: Potential runtime errors due to incorrect assumptions about file paths or data formats.
- Resolution: Included error handling and debugging statements to check file existence and correct data formats before proceeding with further processing.