Semantic Segmentation with Grounded SAM: Comprehensive Guide

```
pip install opencv-python supervision autodistill_grounded_sam
autodistill matplotlib numpy torch
```

Introduction

Semantic segmentation is a powerful computer vision technique that involves partitioning an image into regions based on predefined classes. This guide details the process of performing semantic segmentation using **Grounded SAM** (Segment Anything Model) to identify and quantify specific classes within an image, such as green areas, buildings, and rivers. The workflow includes loading an image, predicting segmentation masks, applying colored overlays, calculating area percentages for target classes, and saving the annotated results.

Prerequisites

Before proceeding, ensure you have the following:

- **Python**: Version 3.7 or higher.
- **pip**: Python package installer.
- **Virtual Environment** (optional but recommended): To manage dependencies.

Python Libraries Required:

- `opencv-python`
- `matplotlib`
- `numpy`
- `torch`
- `torchvision`
- `supervision`
- `autodistill-grounded-sam`

Detailed Workflow

The segmentation process involves several key steps. Below is a detailed explanation of each component within the script.

1. Define Ontology

Purpose: Specifies the classes that the model should detect and segment within the image.

Code:

```
from autodistill.detection import CaptionOntology

# Define your ontology with the remaining classes and 'hedge'
ontology = CaptionOntology({
    "building": "building",
    "grass": "grass",
    "plants": "plants",
    "trees": "trees",
    "hedge": "hedge",
    "river": "river",
    "sky": "sky"
})
```

Explanation: The ontology defines seven classes: building, grass, plants, trees, hedge, river, and sky. This allows the model to focus on these specific entities during segmentation.

2. Initialize the Model

Purpose: Sets up the Grounded SAM model with the defined ontology.

Code:

```
from autodistill_grounded_sam import GroundedSAM

# Initialize Grounded SAM with the updated ontology
base_model = GroundedSAM(ontology=ontology)
```

Explanation: Initializes the Grounded SAM model, enabling it to recognize and segment the classes defined in the ontology.

3. Load the Image

Purpose: Reads the image that will undergo segmentation.

Code:

```
import cv2
import os

# Load your image
image_path = "IMG_4080.png"  # Update this with your image path
image = cv2.imread(image_path)
if image is None:
    raise ValueError(f"Image not found at {image_path}")
```

Explanation: Uses OpenCV to load the image from the specified path. If the image isn't found, the script raises an error.

4. Predict Segmentation Masks

Purpose: Generates segmentation masks for each defined class in the image.

Code:

```
# Predict segmentation masks
detections = base_model.predict(image_path)

# Debugging: Print the detections
print("Detections:", detections)
```

Explanation: The model processes the image and returns detections, which include masks and class IDs for each detected entity.

5. Define Color Map

Purpose: Assigns specific colors to each class for clear visualization in the segmented image.

Code:

```
# Define a color map for each remaining class
# Colors are in RGB format
color_map = {
    "building": (128, 128, 128),   # Grey
    "grass": (0, 255, 0),          # Green
    "plants": (0, 200, 0),         # Medium Green
    "trees": (0, 100, 0),          # Dark Green
    "hedge": (50, 205, 50),        # Lime Green
    "river": (0, 0, 255),          # Blue
    "sky": (128, 0, 128)           # Purple
}
```

Explanation: Each class is assigned a unique RGB color, facilitating easy differentiation when overlays are applied.

6. Map Class IDs to Labels

Purpose: Maps numerical class IDs returned by the model to their corresponding labels.

Code:

```
# Map class IDs to labels based on the updated ontology
id_to_label = [
    "building",   # class_id = 0
    "grass",      # class_id = 1
    "plants",     # class_id = 2
    "trees",      # class_id = 3
    "hedge",      # class_id = 4
    "river",      # class_id = 5
    "sky"         # class_id = 6
]
```

Explanation: Ensures that each detected class ID can be accurately referenced by its label in subsequent steps.

7. Apply Custom Masks

Purpose: Overlays colored masks onto the original image based on detected classes.

Code:

```
import numpy as np

def apply_custom_masks(image, detections, color_map, id_to_label, alpha=0.5):
    """
    Overlays colored masks on the image based on detections.

    :param image: Original image in BGR format.
    :param detections: Detections object containing masks and class_ids.
    :param color_map: Dictionary mapping labels to RGB colors.
    :param id_to_label: List mapping class_id to label.
    :param alpha: Transparency factor for mask overlay.
    :return: Annotated image with colored masks.
    """
    annotated_image = image.copy()
    num_detections = detections.class_id.shape[0]

    for i in range(num_detections):
        class_id = detections.class_id[i]
        if class_id >= len(id_to_label):
            print(f"Warning: class_id {class_id} is out of bounds. Skipping.")
            continue
        label = id_to_label[class_id]

        if label not in color_map:
            print(f"Warning: label '{label}' not in color_map. Skipping.")
            continue

        mask = detections.mask[i]
        color = color_map[label][::-1]  # Convert RGB to BGR for OpenCV

        # Create a colored mask
        colored_mask = np.zeros_like(image, dtype=np.uint8)
        colored_mask[:] = color

        # Ensure mask is of type bool
        mask_bool = mask.astype(bool)

        # Apply the mask with transparency
        annotated_image[mask_bool] = cv2.addWeighted(
            annotated_image, 1 - alpha, colored_mask, alpha, 0
        )[mask_bool]

    return annotated_image
```

Explanation:

- **Overlay Mechanism**: Combines the original image with the colored masks using alpha blending for transparency.
- **Mask Processing**: Converts each mask to a boolean array to apply the overlay accurately.

8. Add Legend

Purpose: Adds a legend to the segmented image, indicating which color corresponds to which class.

Code:

```
def add_legend(image, color_map, position=(10, 30)):
    """
    Adds a legend to the image.

    :param image: Image in BGR format.
    :param color_map: Dictionary mapping labels to RGB colors.
    :param position: Starting position for the legend.
    :return: Image with legend.
    """
    x, y = position
    font = cv2.FONT_HERSHEY_SIMPLEX
    font_scale = 0.6
    font_thickness = 2
    box_height = 20
    box_width = 20
    spacing = 10

    for label, color in color_map.items():
        # Draw color box
        cv2.rectangle(
            image,
            (x, y),
            (x + box_width, y + box_height),
            color[::-1],  # Convert to BGR
            -1
        )
        # Put label text
        cv2.putText(
            image,
            label.replace("_", " ").title(),  # Format label text
            (x + box_width + spacing, y + box_height - 5),
            font,
            font_scale,
            (255, 255, 255),  # White text
            font_thickness,
            cv2.LINE_AA
        )
        y += box_height + spacing  # Move to next line

    return image
```

Explanation:

- **Legend Components**:
    - **Color Box**: Represents the color assigned to a class.
    - **Label Text**: Describes which class the color corresponds to.
- **Positioning**: The legend starts at the `(x, y)` coordinates specified by the `position` parameter and stacks vertically.

9. Calculate and Annotate Percentages

Purpose: Calculates the percentage area of target labels (green, building, river) and annotates these percentages on separate mask images.

Code:

```
def calculate_and_annotate_percentage(image, detections, id_to_label, target_labels,
category_name):
    """
    Calculates the percentage area of target labels and annotates it on the mask image.

    :param image: Original image in BGR format.
    :param detections: Detections object containing masks and class_ids.
    :param id_to_label: List mapping class_id to label.
    :param target_labels: List of labels to calculate percentage for.
    :param category_name: Name of the category (e.g., "Green", "Building", "River").
    :return: Annotated mask image with percentage.
    """
    # Initialize a combined mask for the target labels
    combined_mask = np.zeros((image.shape[0], image.shape[1]), dtype=bool)

    for i in range(len(detections.class_id)):
        class_id = detections.class_id[i]
        if class_id >= len(id_to_label):
            continue
        label = id_to_label[class_id]
        if label in target_labels:
            mask = detections.mask[i]
            combined_mask = combined_mask | mask  # Combine masks using logical OR

    # Calculate the number of target pixels
    target_pixels = np.sum(combined_mask)

    # Calculate the total number of pixels in the image
```

```
total_pixels = image.shape[0] * image.shape[1]

# Calculate the percentage of target area
target_percentage = (target_pixels / total_pixels) * 100

print(f"\nPercentage of {category_name.lower()} in the image: {target_percentage:.2f}%")

# Create a colored mask for visualization
if category_name.lower() == "green":
    color = (0, 255, 0)  # Pure Green
elif category_name.lower() == "building":
    color = (128, 128, 128)  # Grey
elif category_name.lower() == "river":
    color = (0, 0, 255)  # Blue
else:
    color = (255, 255, 255)  # White as default

mask_visual = np.zeros_like(image, dtype=np.uint8)
mask_visual[:] = (0, 0, 0)  # Start with black

mask_visual[combined_mask] = color  # Apply the color to target areas

# Add the percentage text
font = cv2.FONT_HERSHEY_SIMPLEX
font_scale = 1
font_thickness = 2
text = f"{category_name} Area: {target_percentage:.2f}%"
text_size, _ = cv2.getTextSize(text, font, font_scale, font_thickness)
text_x = 10
text_y = text_size[1] + 10

# Draw a semi-transparent rectangle behind the text for better visibility
rectangle_bgr = (0, 0, 0)  # Black rectangle
cv2.rectangle(
    mask_visual,
    (text_x - 5, text_y - text_size[1] - 5),
    (text_x + text_size[0] + 5, text_y + 5),
    rectangle_bgr,
    -1
)

# Add text
cv2.putText(
    mask_visual,
    text,
    (text_x, text_y),
    font,
    font_scale,
    (255, 255, 255),  # White text
    font_thickness,
    cv2.LINE_AA
)

return mask_visual
```

Explanation:

- **Functionality**:
    - **Mask Aggregation**: Combines masks of specified target labels (e.g., all green-related classes) using a logical OR operation to ensure overlapping areas are counted only once.
    - **Percentage Calculation**: Determines what fraction of the total image area is occupied by the target category.
    - **Annotation**: Overlays the calculated percentage on the corresponding mask image for easy reference.
- **Customization**: The function handles different categories (green, building, river) by assigning appropriate colors and texts.

10. Save Output Images

Purpose: Saves the fully segmented image and the annotated percentage mask images to the `output_images` directory.

Code:

```
# Create Output Directory if Not Exists
output_dir = "output_images"
os.makedirs(output_dir, exist_ok=True)

# Apply custom colored masks
annotated_frame = apply_custom_masks(image, detections, color_map, id_to_label, alpha=0.5)

# Add legend to the annotated image
annotated_frame = add_legend(annotated_frame, color_map)

# Save the Annotated Segmented Image with Legend
segmented_image_path = os.path.join(output_dir, "segmented_image.png")
cv2.imwrite(segmented_image_path, annotated_frame)
```

```
print(f"\nSegmented image saved at: {segmented_image_path}")

# Define target labels for each category
categories = {
   "Green": ["grass", "plants", "trees", "hedge"],
   "Building": ["building"],
   "River": ["river"]
}

# Dictionary to hold paths of percentage mask images
percentage_mask_paths = {}

for category_name, target_labels in categories.items():
   mask_visual = calculate_and_annotate_percentage(
      image, detections, id_to_label, target_labels, category_name
   )
   # Define file name based on category
   mask_filename = f"{category_name.lower()}_percentage_mask.png"
   mask_path = os.path.join(output_dir, mask_filename)
   # Save the mask image
   cv2.imwrite(mask_path, mask_visual)
   print(f"{category_name} percentage mask image saved at: {mask_path}")
   # Store the path if needed later
   percentage_mask_paths[category_name] = mask_path
```

Explanation:

- **Directory Management**: Ensures that the `output_images` directory exists. If not, it creates one.
- **Saving Segmented Image**: The fully segmented image with colored masks and legend is saved as `segmented_image.png`.
- **Saving Percentage Masks**: For each category (Green, Building, River), the annotated mask image with percentage is saved separately:
  - `green_percentage_mask.png`
  - `building_percentage_mask.png`
  - `river_percentage_mask.png`

Running the Script

1. Prepare Your Environment

Ensure all prerequisites are met and the necessary libraries are installed as per the Installation section.

2. Place Your Input Image

Ensure that the image you want to process (e.g., `IMG_4080.png`) is located in the project directory or update the `image_path` variable in the script to point to the correct location.

3. Execute the Script

Run the script using Python:

```
python segmentation_script.py
```

Replace `segmentation_script.py` with the actual name of your Python script.

4. Output

Upon successful execution, the script will:

- Save the **segmented image** with colored overlays and a legend in the `output_images/` directory.
- Save **annotated mask images** for each target category (Green, Building, River) with their respective percentage areas.
- Print the percentage values and confirmation messages in the console.

Console Output Example:

```
Detections: <Detections Object>

Unique class IDs detected:
Class ID: 0, Label: building
Class ID: 1, Label: grass
Class ID: 2, Label: plants
Class ID: 3, Label: trees
Class ID: 4, Label: hedge
Class ID: 5, Label: river
Class ID: 6, Label: sky

Percentage of Green in the image: 35.67%
Green percentage mask image saved at: output_images/green_percentage_mask.png

Percentage of Building in the image: 15.23%
Building percentage mask image saved at: output_images/building_percentage_mask.png
```

```
Percentage of River in the image: 10.45%
River percentage mask image saved at: output_images/river_percentage_mask.png

All images have been successfully saved in the 'output_images' directory.
```

Understanding the Outputs

After running the script, you will find the following images in the `output_images/` directory:

1. **segmented_image.png**:
   - **Description**: The original image with colored masks overlaying the detected classes and a legend indicating class-color associations.
   - **Usage**: Provides a visual representation of all detected classes within the image.
2. **green_percentage_mask.png**:
   - **Description**: A mask highlighting all green-related areas (`grass`, `plants`, `trees`, `hedge`) with the green area percentage annotated.
   - **Usage**: Useful for analyzing vegetation coverage within the image.
3. **building_percentage_mask.png**:
   - **Description**: A mask highlighting building areas with the building area percentage annotated.
   - **Usage**: Helps in assessing the extent of built structures within the image.
4. **river_percentage_mask.png**:
   - **Description**: A mask highlighting river areas with the river area percentage annotated.
   - **Usage**: Useful for evaluating the presence and coverage of water bodies.