```
!pip install -q segmentation-models-pytorch
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 154.8/154.8 kB 3.8 MB/s et
a 0:00:0000:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 4.6 MB/s et
a 0:00:000:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 2.5 MB/s et
a 0:00:000:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 8.0 MB/s et
a 0:00:000:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 32.6 MB/s eta
0:00:00:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 7.3 MB/s et
a 0:00:000:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 8.2 MB/s et
a 0:00:000:00:0100:01
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 85.3 MB/s eta
0:00:00:00:0100:01
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of t
he following dependency conflicts.
pylibcugraph-cu12 24.12.0 requires pylibraft-cu12==24.12.*, but you ha
ve pylibraft-cu12 25.2.0 which is incompatible.
pylibcugraph-cu12 24.12.0 requires rmm-cu12==24.12.*, but you have rmm
-cu12 25.2.0 which is incompatible.
```

Library Imports

```
import os
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image
import cv2
from scipy.ndimage import label
from sklearn.metrics import jaccard_score, silhouette_score
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.mixture import GaussianMixture
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from tqdm import tqdm
import albumentations as A
from albumentations.pytorch import ToTensorV2
```

```
import segmentation_models_pytorch as smp
from torchmetrics import JaccardIndex
```

Data pre-processing

```
# Base directory
base_dir = '/kaggle/input/FloodNet-Supervised_v1.0'

# Training data
train_img_dir = '/kaggle/input/floodnet/FloodNet-Supervised_v1.0/train/
train_mask_dir = '/kaggle/input/floodnet/FloodNet-Supervised_v1.0/train

# Validation data
val_img_dir = '/kaggle/input/floodnet/FloodNet-Supervised_v1.0/val/val-
val_mask_dir = '/kaggle/input/floodnet/FloodNet-Supervised_v1.0/val/val
```

Dataset Definition

```
# Loads RGB images and multi-class masks from given directories.
# Applies optional Albumentations transforms during training.

class FloodNetDataset(Dataset):
    def __init__(self, img_dir, mask_dir, transform=None):
        """
        img_dir: Path to RGB images
        mask_dir: Path to grayscale masks (0–9 class labels)
        transform: Optional Albumentations transform
        """
        self.img_dir = img_dir
        self.mask_dir = mask_dir
        self.transform = transform
        self.images = sorted(os.listdir(img_dir))
        self.masks = sorted(os.listdir(mask_dir))

    def __len__(self):
        """Returns total number of samples"""
        return len(self.images)

    def __getitem__(self, idx):
        """
        Returns:
            image (Tensor): RGB image after transform
            mask  (Tensor): Corresponding mask (int64)
        """
        img_path = os.path.join(self.img_dir, self.images[idx])
        mask_path = os.path.join(self.mask_dir, self.masks[idx])
```

```
        image = cv2.imread(img_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE).astype('int6

        if self.transform:
            augmented = self.transform(image=image, mask=mask)
            image = augmented['image']
            mask = augmented['mask']

        return image, torch.as_tensor(mask, dtype=torch.long)
```

```
/usr/local/lib/python3.11/dist-packages/albumentations/__init__.py:28:
UserWarning: A new version of Albumentations is available: '2.0.6' (yo
u have '2.0.4'). Upgrade using: pip install -U albumentations. To disa
ble automatic update checks, set the environment variable NO_ALBUMENTA
TIONS_UPDATE to 1.
  check_for_updates()
```

Image & Mask Transforms

```
# Resizes input to 256×256, normalizes image, and converts both
# image and mask to PyTorch tensors.

transform = A.Compose([
    A.Resize(256, 256),
    A.Normalize(),
    ToTensorV2(transpose_mask=True)
])
```

DataLoaders

```
# Wrap FloodNet datasets in PyTorch DataLoaders for training and valida
# Training loader uses shuffling and larger batch size; validation does

train_dataset = FloodNetDataset(train_img_dir, train_mask_dir, transfor
val_dataset = FloodNetDataset(val_img_dir, val_mask_dir, transform)

train_loader = DataLoader(
    train_dataset,
    batch_size=4,
    shuffle=True,
    num_workers=2,
    pin_memory=True,
    drop_last=True  # Ensure consistent batch size
)

val_loader = DataLoader(
    val_dataset,
```

```
    batch_size=2,
    shuffle=False,
    num_workers=2,
    pin_memory=True,
    drop_last=True
)
```

ᡣ𐭩‿ᡣ𐭩‿⊹ ࣪ ˖★ᡣ𐭩Supervisedᡣ𐭩‿ᡣ𐭩‿⊹ ࣪ ˖★ᡣ𐭩

Set-up the model

```
# Device setup (GPU if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# DeepLabV3+ with ResNet-50 encoder
model = smp.DeepLabV3Plus(
    encoder_name="resnet50",
    encoder_weights="imagenet",
    in_channels=3,
    classes=10,
    activation=None
).to(device)

# Cross-entropy loss for multi-class segmentation
loss_fn = nn.CrossEntropyLoss().to(device)

# Adam optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

# Mean Intersection over Union (mIoU) for multi-class evaluation
iou_metric = JaccardIndex(task="multiclass", num_classes=10).to(device)
```

```
config.json:   0%|          | 0.00/156 [00:00<?, ?B/s]
model.safetensors:   0%|          | 0.00/102M [00:00<?, ?B/s]
```

Training & Validation Loop

```
# Trains model for multiple epochs, evaluates on validation set,
# and saves the model checkpoint with the best IoU.

EPOCHS = 10
best_iou = 0

for epoch in range(EPOCHS):
```

```python
    print(f"\nEpoch {epoch + 1}/{EPOCHS}")

    # ---- Training ----
    model.train()
    train_loss = 0
    train_iou = 0

    for images, masks in tqdm(train_loader):
        images = images.to(device)
        masks = masks.to(device).long()

        optimizer.zero_grad()
        outputs = model(images)

        loss = loss_fn(outputs, masks)
        loss.backward()
        optimizer.step()

        preds = torch.argmax(outputs, dim=1)
        train_loss += loss.item()
        train_iou += iou_metric(preds, masks).item()

    avg_train_loss = train_loss / len(train_loader)
    avg_train_iou = train_iou / len(train_loader)
    print(f"Train Loss: {avg_train_loss:.4f}, IoU: {avg_train_iou:.4f}"
    iou_metric.reset()

    # ---- Validation ----
    model.eval()
    val_loss = 0
    val_iou = 0

    with torch.no_grad():
        for images, masks in val_loader:
            images = images.to(device)
            masks = masks.to(device).long()

            outputs = model(images)
            loss = loss_fn(outputs, masks)

            preds = torch.argmax(outputs, dim=1)
            val_loss += loss.item()
            val_iou += iou_metric(preds, masks).item()

    avg_val_loss = val_loss / len(val_loader)
    avg_val_iou = val_iou / len(val_loader)
    print(f"Val Loss: {avg_val_loss:.4f}, IoU: {avg_val_iou:.4f}")
    iou_metric.reset()

    # ---- Save Best Model ----
```

```python
    if avg_val_iou > best_iou:
        best_iou = avg_val_iou
        torch.save(model.state_dict(), "best_model.pth")
        print("Saved new best model!")
```

```
🔁 Epoch 1/10
100%|██████████| 361/361 [04:03<00:00,  1.48it/s]
✅ Train Loss: 0.9855, IoU: 0.2923
🧪 Val Loss: 0.5723, IoU: 0.4880
💾 Saved new best model!

🔁 Epoch 2/10
100%|██████████| 361/361 [03:16<00:00,  1.84it/s]
✅ Train Loss: 0.5765, IoU: 0.4100
🧪 Val Loss: 0.4428, IoU: 0.5143
💾 Saved new best model!

🔁 Epoch 3/10
100%|██████████| 361/361 [03:16<00:00,  1.84it/s]
✅ Train Loss: 0.4879, IoU: 0.4388
🧪 Val Loss: 0.6038, IoU: 0.4942

🔁 Epoch 4/10
100%|██████████| 361/361 [03:14<00:00,  1.85it/s]
✅ Train Loss: 0.4160, IoU: 0.4810
🧪 Val Loss: 0.4569, IoU: 0.5439
💾 Saved new best model!

🔁 Epoch 5/10
100%|██████████| 361/361 [03:16<00:00,  1.83it/s]
✅ Train Loss: 0.3859, IoU: 0.4879
🧪 Val Loss: 0.4140, IoU: 0.5516
💾 Saved new best model!

🔁 Epoch 6/10
100%|██████████| 361/361 [03:18<00:00,  1.82it/s]
✅ Train Loss: 0.3467, IoU: 0.5186
🧪 Val Loss: 0.4498, IoU: 0.5708
💾 Saved new best model!

🔁 Epoch 7/10
100%|██████████| 361/361 [03:17<00:00,  1.83it/s]
✅ Train Loss: 0.2929, IoU: 0.5600
```

```
🧪 Val Loss: 0.4169, IoU: 0.5816
💾 Saved new best model!

🔄 Epoch 8/10
100%|██████████| 361/361 [03:17<00:00,  1.83it/s]
✅ Train Loss: 0.2809, IoU: 0.5569
🧪 Val Loss: 0.4854, IoU: 0.5714

🔄 Epoch 9/10
100%|██████████| 361/361 [03:17<00:00,  1.83it/s]
✅ Train Loss: 0.2467, IoU: 0.5942
🧪 Val Loss: 0.5021, IoU: 0.5922
💾 Saved new best model!

🔄 Epoch 10/10
100%|██████████| 361/361 [03:17<00:00,  1.83it/s]
✅ Train Loss: 0.2339, IoU: 0.6009
🧪 Val Loss: 0.4899, IoU: 0.5877
```

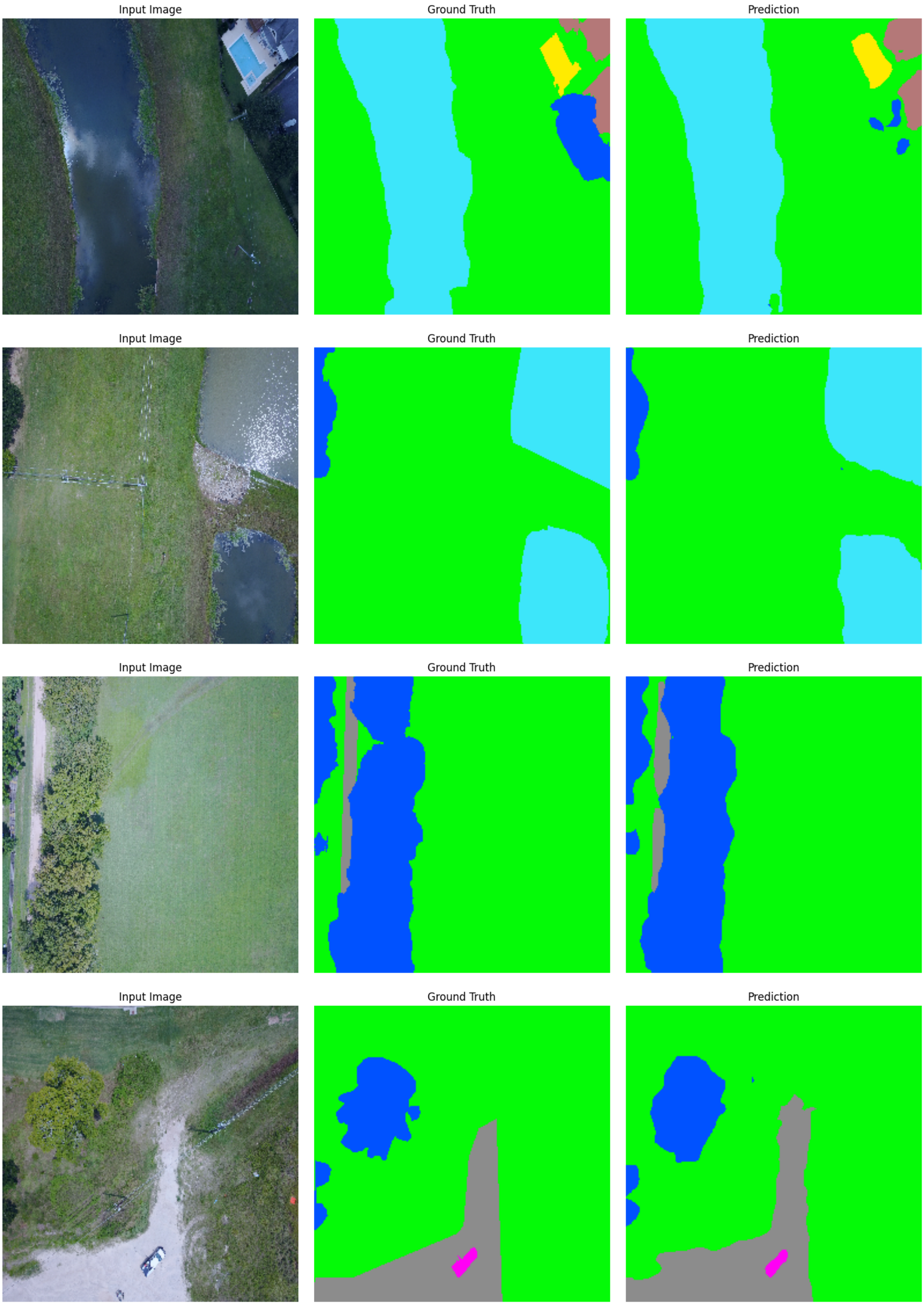
Visualization

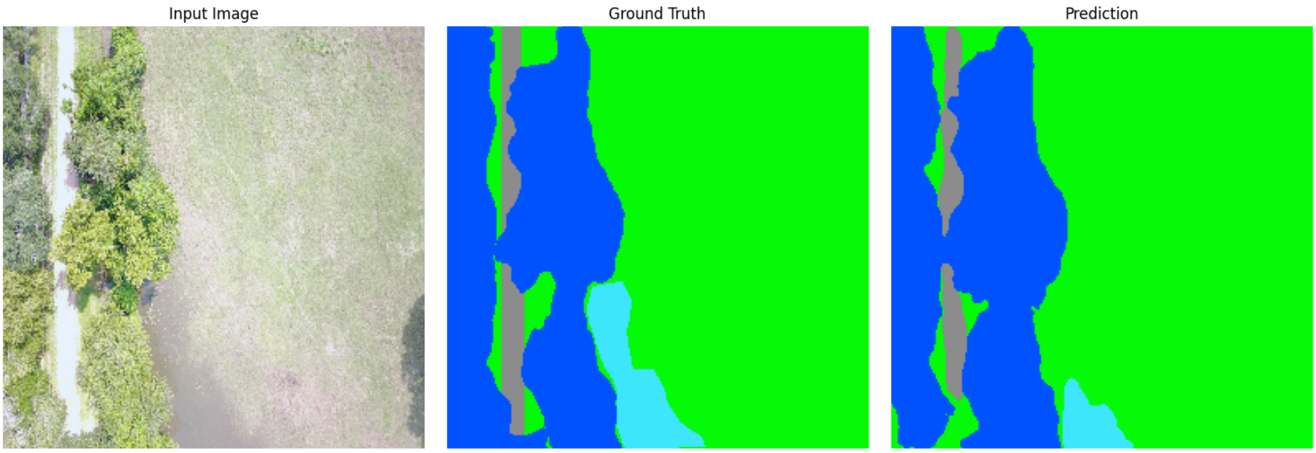
Decode Segmentation Mask

```
# Converts a single-channel mask (class indices 0–9) to an RGB image
# using predefined color mappings for visualization.

def decode_segmap(mask):
    colors = np.array([
        [0, 0, 0],          # 0: background
        [255, 0, 0],        # 1: flooded building
        [180, 120, 120],    # 2: non-flooded building
        [160, 150, 20],     # 3: flooded road
        [140, 140, 140],    # 4: non-flooded road
        [61, 230, 250],     # 5: water
        [0, 82, 255],       # 6: tree
        [255, 0, 245],      # 7: vehicle
        [255, 235, 0],      # 8: pool
        [4, 250, 7],        # 9: grass
    ], dtype=np.uint8)

    rgb_mask = colors[mask]
    return rgb_mask
```

Visualize Sample Predictions from Model

```
all_images, all_masks = [], []
for i, (images, masks) in enumerate(val_loader):
    all_images.append(images)
    all_masks.append(masks)

all_images = torch.cat(all_images, dim=0)
all_masks = torch.cat(all_masks, dim=0)

num_images_needed = 5
random_indices = random.sample(range(all_images.size(0)), num_images_ne

images = all_images[random_indices].to(device)
masks = all_masks[random_indices]

with torch.no_grad():
    outputs = model(images)
    preds = torch.argmax(outputs, dim=1).cpu().numpy()

for i in range(num_images_needed):
    img = images[i].cpu().permute(1, 2, 0).numpy()
    img = (img - img.min()) / (img.max() - img.min())

    gt = masks[i].cpu().numpy()
    pr = preds[i]

    fig, axs = plt.subplots(1, 3, figsize=(15, 5))
    axs[0].imshow(img)
    axs[0].set_title("Input Image")

    axs[1].imshow(decode_segmap(gt))
    axs[1].set_title("Ground Truth")

    axs[2].imshow(decode_segmap(pr))
    axs[2].set_title("Prediction")

    for ax in axs:
        ax.axis("off")
    plt.tight_layout()
    plt.show()
```


Input Image
Ground Truth
Prediction
Input Image
Ground Truth
Prediction
Input Image
Ground Truth
Prediction
Input Image
Ground Truth
Prediction



ᡣ𐭩ᡣ𐭩 ⁺₊★ ᡣ𐭩unsupervisedᡣ𐭩ᡣ𐭩 ⁺₊★ ᡣ𐭩

Save Predicted Masks Containing Flood Classes (1 or 3 Only)

```
# Converts model predictions to binary flood masks, filters those
# containing flood-related classes, and saves them as PNGs.

save_dir = "/kaggle/working/predictedFLOOD_masksV"
os.makedirs(save_dir, exist_ok=True)

# Define target classes related to flood
FLOOD_CLASSES = [1, 3]

def convert_to_binary_mask(mask):
    """
    Converts multi-class mask to binary mask (1 if in FLOOD_CLASSES, el
    """
    return np.isin(mask, FLOOD_CLASSES).astype(np.uint8)

# Set model to evaluation mode
model.eval()
original_filenames = []

# Loop over validation set and collect predictions
with torch.no_grad():
    for batch_idx, (images, masks) in enumerate(val_loader):
        images = images.to(device)
        outputs = model(images)
        preds = torch.argmax(outputs, dim=1).cpu().numpy()
```

```
        for i in range(preds.shape[0]):
            pred_mask = preds[i].astype(np.uint8)
            binary_mask = convert_to_binary_mask(pred_mask)
            has_flood = np.any(binary_mask == 1)

            if has_flood:
                pred_name = f"pred_{batch_idx}_{i}.png"
                save_path = os.path.join(save_dir, pred_name)
                Image.fromarray(pred_mask).save(save_path)

                # Store reference to prediction and corresponding image
                original_filenames.append({
                    "filename": pred_name,
                    "original_image": f"val_img_{batch_idx}_{i}.jpg"  #
                })
```

Extract Flood Features from Predicted Masks

```
# Calculates per-mask statistics

mask_dir = "/kaggle/working/predictedFLOOD_masksV"
#PIXEL_AREA_M2 = 0.000225 Each pixel covers 1.5cm x 1.5cm

features = []

def get_flood_spread_index(binary_mask):
    """
    Returns number of connected flood regions in the binary mask.
    """
    structure = np.ones((3, 3), dtype=int)
    labeled, num_components = label(binary_mask, structure=structure)
    return num_components

def get_largest_component_size(binary_mask):
    """
    Returns size (in pixels) of the largest flood component.
    Ignores background (label 0).
    """
    structure = np.ones((3, 3), dtype=int)
    labeled, num_components = label(binary_mask, structure=structure)
    if num_components == 0:
        return 0
    return np.max(np.bincount(labeled.ravel())[1:])  # Skip background

# Iterate through saved predicted masks
for fname in os.listdir(mask_dir):
    if not fname.endswith(".png"):
        continue
```

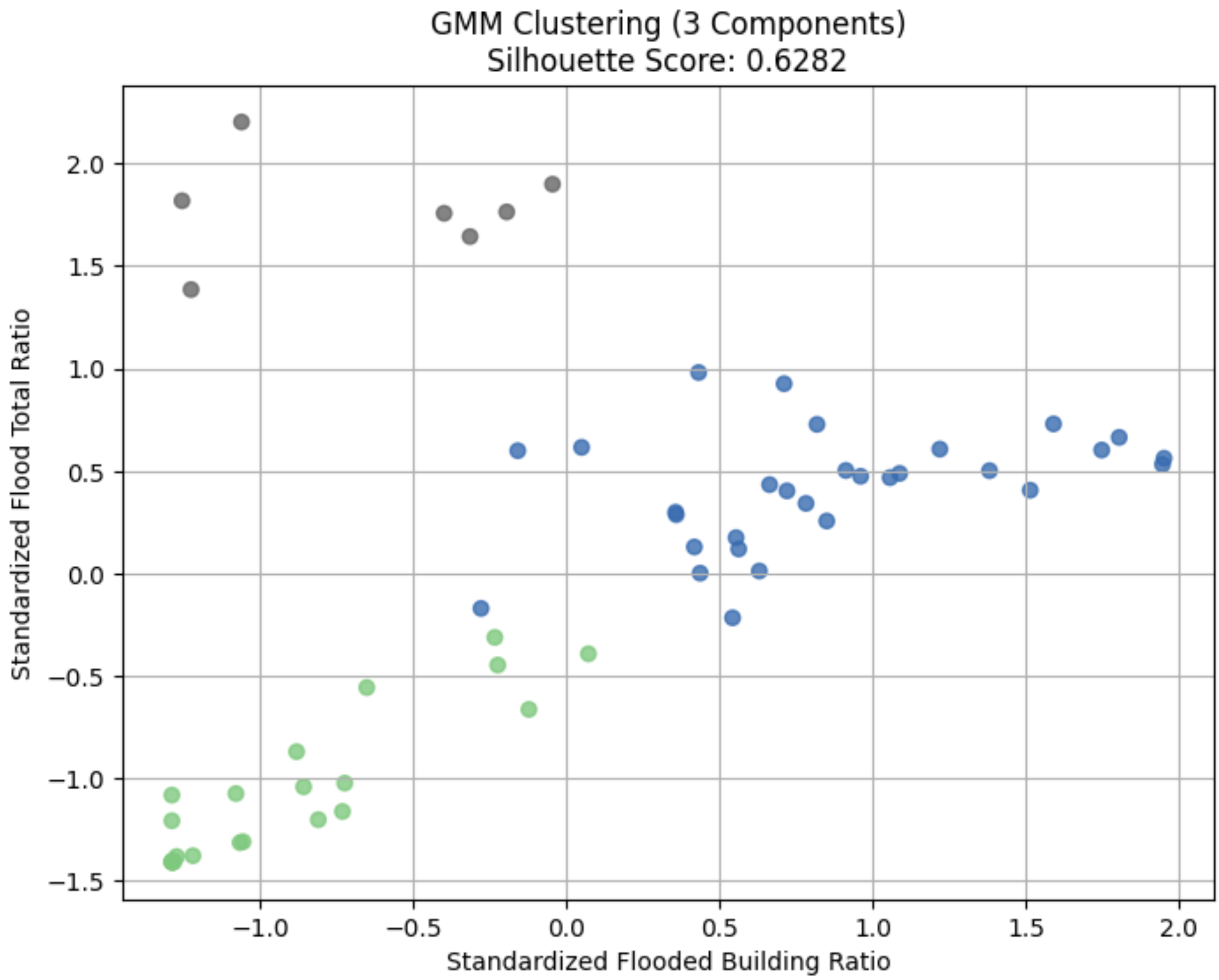
```
    mask_path = os.path.join(mask_dir, fname)
    mask = np.array(Image.open(mask_path))

    # Create binary mask where 1,and 3 are flood-related classes
    binary_mask = ((mask == 1) | (mask == 3)).astype(np.uint8)
    total_pixels = mask.shape[0] * mask.shape[1]

    flooded_area_px = np.sum(binary_mask)
    largest_component_px = get_largest_component_size(binary_mask)

    # Collect desired features
    feature = {
        "filename": fname,
        "flooded_building_ratio": np.sum(mask == 1) / total_pixels,
        "flood_total_ratio": flooded_area_px / total_pixels,
    }

    features.append(feature)

# Convert features to DataFrame
features_df = pd.DataFrame(features)
```

GMM

```
# Applies Gaussian Mixture Model to identify 3 distinct flood profiles
# based on building and total flood ratios. Evaluates clustering qualit
# using Silhouette Score.

# Extract input features
X = features_df[['flooded_building_ratio', 'flood_total_ratio']].values

# Normalize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Fit GMM with 3 components
gmm = GaussianMixture(n_components=3, random_state=42)
gmm_labels = gmm.fit_predict(X_scaled)

# Store cluster labels in DataFrame
features_df['GMM_cluster'] = gmm_labels

# Evaluate clustering quality
gmm_silhouette = silhouette_score(X_scaled, gmm_labels)
print(f"Silhouette Score for GMM clustering: {gmm_silhouette:.4f}")

# Visualize clustered points
plt.figure(figsize=(8, 6))
```

```
scatter = plt.scatter(
    X_scaled[:, 0],
    X_scaled[:, 1],
    c=gmm_labels,
    cmap='Accent',
    alpha=0.8
)

plt.title(f"GMM Clustering (3 Components)\nSilhouette Score: {gmm_silho
plt.xlabel("Standardized Flooded Building Ratio")
plt.ylabel("Standardized Flood Total Ratio")
plt.grid(True)
plt.show()
```

```
Silhouette Score for GMM clustering: 0.6282
```



Analyze Feature Averages Per GMM Cluster

```
# Computes the mean values of key flood features for each cluster.

for i in range(3):
    # Get indices of samples belonging to cluster i
    cluster_indices = np.where(gmm_labels == i)[0]
```

```
    # Extract corresponding feature rows
    cluster_features = features_df.iloc[cluster_indices][['flooded_buil

    # Print average values per cluster
    print(f"Cluster {i}")
    print(cluster_features.mean())
    print("-" * 30)
```

```
Cluster 0
flooded_building_ratio    0.039680
flood_total_ratio         0.088212
dtype: float64
------------------------------
Cluster 1
flooded_building_ratio    0.241860
flood_total_ratio         0.473856
dtype: float64
------------------------------
Cluster 2
flooded_building_ratio    0.072534
flood_total_ratio         0.832162
dtype: float64
------------------------------
```

Map GMM Cluster to Risk Labels based on the analysis

```
# Assigns semantic flood risk levels to each cluster ID.

cluster_risk_map = {
    0: "Low Risk",         # Typically low flood presence
    2: "Moderate Risk",    # Moderate flood or partial building impact
    1: "Severe Risk"       # High flood spread, especially on buildings
}
```