Create Your Own Private Subset Dataset for Cervical Spine Fracture Detection (Kaggle Notebook)

Before you get started, double-check you have the datasets loaded as Input (check right-hand side). You want to make sure you have these datasets selected:

- rsna-2022-spine-fracture-detection-metadata
- RSNA 2022 Cervical Spine Fracture Detection

```
# This will add the pydicom decompressors
!pip install pydicom pylibjpeg pylibjpeg-libjpeg
# Re-install a compatible version of numpy
!pip install numpy==1.26.4
```

!!STOP AND RESTART SESSION!!

YOU ONLY NEED TO DO THIS ONCE

This makes sure the above dependencies are installed, and we're able to decompress images.

You can skip the first two cells after restarting.

```
# Import libraries
import pandas as pd
import numpy as np
import pydicom
import h5py
import cv2
import os
from tqdm.auto import tqdm
import warnings
import json
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GroupShuffleSplit
import matplotlib.pyplot as plt
import seaborn as sns

random_state = 24

# Suppress the DeprecationWarning
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

```
# Set paths to the Kaggle data
bbox_csv_path = '/kaggle/input/rsna-2022-cervical-spine-fracture-detection/train_bounding_boxes.csv'

# We're using the clean version from the metadata dataset
meta_csv_path = '/kaggle/input/rsna-2022-spine-fracture-detection-metadata/meta_train_clean.csv'

# This is the path to all 136GB of images
image_data_path = '/kaggle/input/rsna-2022-cervical-spine-fracture-detection/train_images/'

# Set the output path
hdf5_save_path = '/kaggle/working/fracture_dataset_subset.h5'

# Settings
image_size = 256
```

```
negative_ratio = 3 # for each positive, we add 3 negative samples
max_boxes = 10
```

```
# Load and view metadata
meta_df = pd.read_csv(meta_csv_path)
meta_df.head(3)
```

	StudyInstanceUID	Slice	ImageHeight	ImageWidth	SliceThickness	ImagePositionPatient_x	ImagePositionPat
0	1.2.826.0.1.3680043.10001	1	512	512	0.625	-52.308	-
1	1.2.826.0.1.3680043.10001	2	512	512	0.625	-52.308	-
2	1.2.826.0.1.3680043.10001	3	512	512	0.625	-52.308	-

```
# Load and view bounding box data
bbox_df = pd.read_csv(bbox_csv_path)
bbox_df.head(3)
```

	StudyInstanceUID	x	y	width	height	slice_number
0	1.2.826.0.1.3680043.10051	219.27715	216.71419	17.30440	20.38517	133
1	1.2.826.0.1.3680043.10051	221.56460	216.71419	17.87844	25.24362	134
2	1.2.826.0.1.3680043.10051	216.82151	221.62546	27.00959	26.37454	135

```
# Load and merge metadata

# Rename slice columns to match
meta_df = meta_df.rename(columns={'Slice': 'SliceNumber'})
bbox_df = bbox_df.rename(columns={'slice_number': 'SliceNumber'})

# Group by slice and aggregate bboxes into a list [x, y, width, height]
print('Aggregating bounding boxes per slice...')
bbox_grouped = bbox_df.groupby(['StudyInstanceUID', 'SliceNumber'])[['x', 'y', 'width', 'height']].apply(
    lambda x: x.values.tolist()
).reset_index(name='bboxes')
print(f"Found {len(bbox_grouped)} slices with bounding boxes.")

# Merge all slices with the bounding box data
all_slices_df = pd.merge(
    meta_df,
    bbox_grouped,
    on=['StudyInstanceUID', 'SliceNumber'],
    how='left'
)

# Create final labels
# 'is_positive' is 1 if 'bboxes' is not NaN, 0 otherwise
all_slices_df['is_positive'] = all_slices_df['bboxes'].notna().astype(int)

# Fill NaN in 'bboxes' with an empty list for consistency
all_slices_df['bboxes'] = all_slices_df['bboxes'].apply(
    lambda x: x if isinstance(x, list) else []
)

print('Metadata merge complete.')
```

```
Aggregating bounding boxes per slice...
Found 7217 slices with bounding boxes.
Metadata merge complete.
```

```
# Check image sizes
print(all_slices_df['ImageHeight'].value_counts())
print('\n',all_slices_df['ImageWidth'].value_counts())
```

```
ImageHeight
512    710819
768       782
Name: count, dtype: int64

 ImageWidth
512    710574
768       782
519       245
Name: count, dtype: int64
```

```
# Filter for 512x512 images only
all_slices_df = all_slices_df[
    (all_slices_df['ImageHeight'] == 512) &
    (all_slices_df['ImageWidth'] == 512)
]
print(all_slices_df['ImageHeight'].value_counts())
print('\n',all_slices_df['ImageWidth'].value_counts())
```

```
ImageHeight
512    710574
Name: count, dtype: int64

 ImageWidth
512    710574
Name: count, dtype: int64
```

```
# Create patient-level splits

# Define Split Ratios
# 70% Train, 15% Validation, 15% Test
train_size = 0.70
val_size = 0.15
test_size = 0.15

# Prepare Patient-Level Data
# One row per patient for splitting
patient_df = all_slices_df.groupby('StudyInstanceUID').agg({
    'is_positive': 'max', # If patient has any fracture slice, they are positive
    'SliceNumber': 'count' # Count slices per patient
}).reset_index()

print(f'Total Patients: {len(patient_df)}')
print(f"Positive Patients: {patient_df['is_positive'].sum()}")
```

```
Total Patients: 2016
Positive Patients: 234
```

```
# First split: train / (val + test)

# We use GroupShuffleSplit to split by patient id while keeping class balance
splitter = GroupShuffleSplit(test_size=(val_size + test_size), n_splits=1, random_state=42)
train_idxs, temp_idxs = next(splitter.split(patient_df, y=patient_df['is_positive'], groups=patient_df['StudyI
train_patients = patient_df.iloc[train_idxs]
temp_patients = patient_df.iloc[temp_idxs]

# Second Split: val and test
# Split the remaining 30% into two equal halves (15% Val, 15% Test)
splitter_2 = GroupShuffleSplit(test_size=0.5, n_splits=1, random_state=42)
val_idxs, test_idxs = next(splitter_2.split(temp_patients, y=temp_patients['is_positive'], groups=temp_patient

val_patients = temp_patients.iloc[val_idxs]
test_patients = temp_patients.iloc[test_idxs]

print(f'Train Patient Count: {len(train_patients)}')
print(f'Validation Patient Count: {len(val_patients)}')
print(f'Test Patient Count: {len(test_patients)}')
```

```
Train Patient Count: 1411
Validation Patient Count: 302
```

```
Test Patient Count: 303
```

```python
# Map splits back to slice dataframe based on patient id
# 0 = Train, 1 = Val, 2 = Test
train_uids = set(train_patients['StudyInstanceUID'])
val_uids = set(val_patients['StudyInstanceUID'])
test_uids = set(test_patients['StudyInstanceUID'])

def assign_split(uid):
    if uid in train_uids: return 0
    if uid in val_uids: return 1
    if uid in test_uids: return 2

all_slices_df['split'] = all_slices_df['StudyInstanceUID'].apply(assign_split)
print('Split column created.')
```

```
Split column created.
```

```python
# Create the Final Subset DataFrame

# Separate positives and negatives
positive_slices = all_slices_df[all_slices_df['is_positive'] == 1]
negative_slices = all_slices_df[all_slices_df['is_positive'] == 0]

# Sample negatives
negative_subset_list = []
for split_id in [0, 1, 2]:
    split_negatives = negative_slices[negative_slices['split'] == split_id]
    split_positives = positive_slices[positive_slices['split'] == split_id]

    # Use our specified ratio
    n_neg = int(len(split_positives) * negative_ratio)
    n_neg = min(n_neg, len(split_negatives))
    negative_subset_list.append(split_negatives.sample(n=n_neg, random_state=42))

negative_subset = pd.concat(negative_subset_list)

# Combine everything
final_subset_df = pd.concat([positive_slices, negative_subset]).sample(frac=1, random_state=42).reset_index(dr
final_subset_df.head()
```

	StudyInstanceUID	SliceNumber	ImageHeight	ImageWidth	SliceThickness	ImagePositionPatient_x	ImagePosit
0	1.2.826.0.1.3680043.25600	155	512	512	0.625	-71.000000	
1	1.2.826.0.1.3680043.24045	281	512	512	0.500	-115.009300	
2	1.2.826.0.1.3680043.14348	352	512	512	0.600	-46.832031	
3	1.2.826.0.1.3680043.12632	405	512	512	0.625	-64.000000	
4	1.2.826.0.1.3680043.14087	364	512	512	0.500	-79.921700	

```python
# Verification
print(f"Train Slices: {len(final_subset_df[final_subset_df['split']==0])}")
print(f"Val Slices:   {len(final_subset_df[final_subset_df['split']==1])}")
print(f"Test Slices:  {len(final_subset_df[final_subset_df['split']==2])}")

# Check for leakage
train_uids = set(final_subset_df[final_subset_df['split']==0]['StudyInstanceUID'])
val_uids   = set(final_subset_df[final_subset_df['split']==1]['StudyInstanceUID'])
test_uids  = set(final_subset_df[final_subset_df['split']==2]['StudyInstanceUID'])

if len(train_uids.intersection(val_uids)) == 0 and len(train_uids.intersection(test_uids)) == 0:
    print('✅ SUCCESS: No patient leakage detected!')
else:
    print('❌ WARNING: Leakage detected!')
```

```
Train Slices: 20744
Val Slices:   4180
Test Slices:  3888
✅ SUCCESS: No patient leakage detected!
```

```
  1 # View boxes before box scaling
  2 final_subset_df['bboxes']
```

```
0                                                      []
1                                                      []
2                                                      []
3                                                      []
4                                                      []
                              ...
28807                                                  []
28808                          [[257.0, 160.0, 126.0, 95.0]]
28809    [[235.76963, 221.0, 84.23037, 83.94740999999999]]
28810                                                  []
28811                                                  []
Name: bboxes, Length: 28812, dtype: object
```

```
  1 # Scale bounding boxes from original 512x512 to 256x256
  2 # We can divide each box coordinate by 2, or multiply by half
  3 scale_factor = 0.5
  4
  5 def scale_bboxes(bbox_list, scale_factor):
  6     if not bbox_list:  # for negative sample
  7         return []
  8
  9     scaled_boxes = []
 10     for box in bbox_list:
 11         # Scale each coordinate [x, y, w, h]
 12         scaled_box = [coord * scale_factor for coord in box]
 13         scaled_boxes.append(scaled_box)
 14
 15     return scaled_boxes
 16
 17 # Apply this function to the 'bboxes' column
 18 final_subset_df['bboxes'] = final_subset_df['bboxes'].apply(
 19     lambda x: scale_bboxes(x, scale_factor)
 20 )
 21
 22 print('Bounding boxes successfully scaled down for 256x256 size.')
 23 final_subset_df['bboxes'] # after box scaling
```

```
Bounding boxes successfully scaled down for 256x256 size.
0                                                      []
1                                                      []
2                                                      []
3                                                      []
4                                                      []
                              ...
28807                                                  []
28808                           [[128.5, 80.0, 63.0, 47.5]]
28809    [[117.884815, 110.5, 42.115185, 41.97370499999...
28810                                                  []
28811                                                  []
Name: bboxes, Length: 28812, dtype: object
```

```
  1 # Image preprocessing function
  2 def load_and_process_dicom(uid, slice_num, target_size):
  3     # Build the local file path
  4     file_path = f'{image_data_path}/{uid}/{slice_num}.dcm'
  5
  6     ds = pydicom.dcmread(file_path)
  7
  8     # Get pixel array
  9     img = ds.pixel_array.astype(np.float32)
 10
 11     # Perform min-max normalization (scales pixel intensity to range [0,1])
 12     img_min = np.min(img)
```

```
    img_max = np.max(img)
    if img_max > img_min:
        img = (img - img_min) / (img_max - img_min)
    else:
        img = np.zeros(img.shape) # Handle black images

    # Resize
    img = cv2.resize(img, (target_size, target_size), interpolation=cv2.INTER_LINEAR)

    return img
```

```
## This cell was made using Google Gemini AI ##

num_samples = len(final_subset_df)
print(f"Creating HDF5 file at {hdf5_save_path} with {num_samples} total samples...")

# We'll write in chunks of 32 images at a time
image_chunk = (32, image_size, image_size)
bbox_chunk = (32, max_boxes, 4)
text_chunk = (512,) # A good chunk size for 1D arrays

with h5py.File(hdf5_save_path, 'w') as hf:

    # --- Create datasets ---
    dset_images = hf.create_dataset('images', shape=(num_samples, image_size, image_size), dtype='f4', chunks=
    dset_labels = hf.create_dataset('labels', shape=(num_samples,), dtype='i1', chunks=text_chunk)
    dset_bboxes = hf.create_dataset('bboxes', shape=(num_samples, max_boxes, 4), dtype='f4', fillvalue=-1.0, c
    dt_str = h5py.special_dtype(vlen=str)
    dset_uid = hf.create_dataset('StudyInstanceUID', (num_samples,), dtype=dt_str, chunks=text_chunk)
    dset_slice = hf.create_dataset('SliceNumber', (num_samples,), dtype=dt_str, chunks=text_chunk)
    dset_split = hf.create_dataset('split', shape=(num_samples,), dtype='i1', chunks=text_chunk)

    # --- Start the processing loop ---
    print("Starting processing loop...")

    for idx, row in tqdm(final_subset_df.iterrows(), total=num_samples, desc="Processing slices"):
        try:
            # 1. Load and process the image from the local disk
            img = load_and_process_dicom(
                row['StudyInstanceUID'],
                row['SliceNumber'],
                image_size
            )

            # 2. Save data to HDF5 file
            dset_images[idx] = img
            dset_labels[idx] = row['is_positive']
            dset_uid[idx] = row['StudyInstanceUID']
            dset_slice[idx] = str(row['SliceNumber'])
            dset_split[idx] = row['split']

            # 3. Process and save bounding boxes
            bboxes = row['bboxes']
            num_boxes = min(len(bboxes), max_boxes)

            if num_boxes > 0:
                dset_bboxes[idx, :num_boxes, :] = np.array(bboxes[:num_boxes])

        except Exception as e:
            # This will catch any corrupted files
            print(f"\n[Warning] Failed to process slice {row['StudyInstanceUID']}/{row['SliceNumber']}: {e}")

print("\n--- HDF5 file creation complete! ---")
print(f"File saved to: {hdf5_save_path}")
!ls -lh /kaggle/working/
```

```
Creating HDF5 file at /kaggle/working/fracture_dataset_subset.h5 with 28812 total samples...
Starting processing loop...
Processing slices:   0%|          | 0/28812 [00:00<?, ?it/s]

--- HDF5 file creation complete! ---
File saved to: /kaggle/working/fracture_dataset_subset.h5
total 7.1G
-rw-r--r-- 1 root root  113 Nov 26 23:23 dataset-metadata.json
-rw-r--r-- 1 root root 7.1G Nov 27 00:24 fracture_dataset_subset.h5
```

```
# Count total postive and negative samples
with h5py.File(hdf5_save_path, 'r') as f:
    labels = f['labels'][:]
    total_negative = np.sum(labels == 0)
    total_positive = np.sum(labels == 1)

print(f'Total Negative Samples: {total_negative}')
print(f'Total Postiive Samples: {total_positive}')
```

```
Total Negative Samples: 21609
Total Postiive Samples: 7203
```

🔑 Workaround: Creating a Private Dataset with `kaggle.json`

If you are experiencing issues with the Secret add-on feature in Kaggle Notebooks, you can use a **private Kaggle Dataset** to securely store and access your `kaggle.json` file.

Steps to Create the Dataset:

1. **Locate Your File:** Find the `kaggle.json` file you just downloaded on your computer.
2. **Navigate to Dataset Creation:** Go to the Kaggle website and click **"Create"** in the top navigation bar, then select **"New Dataset"**.
3. **Upload the File:**
   - Click **"Upload File"**.
   - Select and upload your `kaggle.json` file.
4. **Configure Dataset Settings:**
   - **Title:** Give your dataset a descriptive name, like `Kaggle API Key`.
   - **Visibility: CRITICALLY IMPORTANT:** Ensure the visibility is set to **"Private"**. This keeps your API key secure.
5. **Create and Finalize:** Click the **"Create"** or **"Publish"** button to finalize the dataset creation.

Steps to Use the Dataset in Your Notebook:

1. **Open Your Notebook:** Open the Kaggle Notebook where you need to use the API key.
2. **Add the Data:** In the right-hand panel of the notebook editor, click on **"+ Add data"**.
3. **Search and Select:** Search for the private dataset you just created (e.g., `Kaggle API Key`) under the **"Your Datasets"** tab and select it.
4. **Access the Key:** Once the dataset is added, your `kaggle.json` file will be accessible in the notebook's file system, typically under the path:

   ```
   /kaggle/input/<your-dataset-name>/kaggle.json
   ```

   You can then modify your script to load the credentials from this path instead of relying on the Secret add-on.

```
# Load API credentials from your private api key dataset
# If you do not have this private dataset setup, see project README for instructions
# Doing this to work around Secret add-ons not working

private_secret_dataset_name = 'DATASET_NAME' ### REPLACE WITH YOUR PRIVATE DATASET NAME ###

CREDENTIALS_PATH = f'/kaggle/input/{private_secret_dataset_name}/kaggle.json'
```

```
# Create the hidden .kaggle directory
!mkdir -p ~/.kaggle

# Copy your key from the private dataset to the correct location
!cp '{CREDENTIALS_PATH}' ~/.kaggle/kaggle.json

# Set the correct permissions for the file
!chmod 600 ~/.kaggle/kaggle.json

print('Kaggle API credentials are now in place.')
```

```
Kaggle API credentials are now in place.
```

```
# Create the private Kaggle dataset

kaggle_username = 'KAGGLE_USERNAME' ### Set KAGGLE_USERNAME ###


# Define your dataset metadata
dataset_metadata = {
  "title": "RSNA 2022 HDF5 Subset",
  "id": f"{kaggle_username}/rsna-2022-hdf5-subset",
  "licenses": [
    {
      "name": "CC0-1.0"
    }
  ]
}

# Write the metadata file
with open('/kaggle/working/dataset-metadata.json', 'w') as f:
    json.dump(dataset_metadata, f)

# Run the create command
# You may have to wait a few minutes for the dataset to fully load to the Kaggle API system
print('Starting dataset creation... This will upload 7.05 GB.')
!kaggle datasets create -p /kaggle/working/ -r zip
```

```
Starting dataset creation... This will upload 7.05 GB.
Starting upload for file fracture_dataset_subset.h5
100%|████████████████████████████████████| 7.05G/7.05G [02:53<00:00, 43.7MB/s]
Upload successful: fracture_dataset_subset.h5 (7GB)
Starting upload for file .virtual_documents.zip
100%|████████████████████████████████████████| 22.0/22.0 [00:00<00:00, 46.9B/s]
Upload successful: .virtual_documents.zip (22B)
Your private Dataset is being created. Please check progress at https://www.kaggle.com/datasets/andymalinsky/rsna-2
```