DICOM Conversion Script (di_convert.py)

```
import os

import pydicom

import numpy as np

from PIL import Image

import re


def get_next_index_tb(target_folder, prefix="TB"):

   existing = [f for f in os.listdir(target_folder) if f.lower().endswith(".png") and
f.startswith(prefix)]

   numbers = [int(re.findall(rf"{prefix}(\d+)\.png", f)[0]) for f in existing if
re.findall(rf"{prefix}(\d+)\.png", f)]

   return max(numbers, default=0) + 1


def convert_dicoms_to_png(source_folder, target_tb_folder, prefix="TB-"):

   os.makedirs(target_tb_folder, exist_ok=True)

   index = get_next_index_tb(target_tb_folder, prefix=prefix)


   for filename in sorted(os.listdir(source_folder)):

      if filename.lower().endswith((".dcm", ".dicom")):

         dicom_path = os.path.join(source_folder, filename)

         output_filename = f"{prefix}{index:05d}.png"

         output_path = os.path.join(target_tb_folder, output_filename)


         try:

            ds = pydicom.dcmread(dicom_path)
```

```
            pixel_array = ds.pixel_array.astype(np.float32)

            # Apply rescale slope/intercept if present
            intercept = ds.get("RescaleIntercept", 0.0)
            slope = ds.get("RescaleSlope", 1.0)
            pixel_array = pixel_array * slope + intercept

            # Normalize to 0–255
            pixel_array -= pixel_array.min()
            pixel_array /= pixel_array.max()
            pixel_array = (pixel_array * 255).astype(np.uint8)

            # Convert grayscale to RGB if needed
            if len(pixel_array.shape) == 2:
                pixel_array = np.stack([pixel_array] * 3, axis=-1)

            Image.fromarray(pixel_array).save(output_path)
            print(f"Saved {output_filename}")
            index += 1
        except Exception as e:
            print(f"Failed to convert {filename}: {e}")

if __name__ == "__main__":
    # Example usage: loop over multiple folders
    source_folders_tb = ["intbtr251","intbtr252","intbtr253","intbtr254",
"intbtr255","intbtr256","intbtr260","intbtr261"]  # Add more as needed
```

```
    target_tb_folder = os.path.join("TB_Chest_Radiography_Database_raw", "TB")

    for folder in source_folders_tb:
        print(f"🔍 Processing folder: {folder}")
        convert_dicoms_to_png(folder, target_tb_folder, prefix="TB-")

print("TB Conversion complete!")

print("Starting Normal dicom files conversion to PNG...")

def get_next_index_norm(target_folder, prefix="Normal"):
    existing = [f for f in os.listdir(target_folder) if f.lower().endswith(".png") and
f.startswith(prefix)]
    numbers = [int(re.findall(rf"{prefix}(\d+)\.png", f)[0]) for f in existing if
re.findall(rf"{prefix}(\d+)\.png", f)]
    return max(numbers, default=0) + 1

def convert_dicoms_to_png(source_folder, target_norm_folder, prefix="Normal-"):
    os.makedirs(target_norm_folder, exist_ok=True)
    index = get_next_index_norm(target_norm_folder, prefix=prefix)

    for filename in sorted(os.listdir(source_folder)):
        if filename.lower().endswith((".dcm", ".dicom")):
            dicom_path = os.path.join(source_folder, filename)
            output_filename = f"{prefix}{index:05d}.png"
            output_path = os.path.join(target_norm_folder, output_filename)
```

```
        try:
            ds = pydicom.dcmread(dicom_path)
            pixel_array = ds.pixel_array.astype(np.float32)

            # Apply rescale slope/intercept if present
            intercept = ds.get("RescaleIntercept", 0.0)
            slope = ds.get("RescaleSlope", 1.0)
            pixel_array = pixel_array * slope + intercept

            # Normalize to 0–255
            pixel_array -= pixel_array.min()
            pixel_array /= pixel_array.max()
            pixel_array = (pixel_array * 255).astype(np.uint8)

            # Convert grayscale to RGB if needed
            if len(pixel_array.shape) == 2:
                pixel_array = np.stack([pixel_array] * 3, axis=-1)

            Image.fromarray(pixel_array).save(output_path)
            print(f"Saved {output_filename}")
            index += 1
        except Exception as e:
            print(f"Failed to convert {filename}: {e}")

if __name__ == "__main__":
```

```
# Example usage: loop over multiple folders
source_folders_norm = ["intbtr12","intbtr13","intbtr29","intbtr65","intbtr67","intbtr78","intbtr79","intbtr81","intbtr103","intbtr104"]  # Add more as needed
target_norm_folder = os.path.join("TB_Chest_Radiography_Database_raw", "Normal")

for folder in source_folders_norm:
    print(f"🔍 Processing folder: {folder}")
    convert_dicoms_to_png(folder, target_norm_folder, prefix="Normal-")
```

Script for Data Splitting (data_split.py)

```
import os

import shutil

import random

import tensorflow as tf

from tqdm import tqdm


# --- Config ---

ORIGINAL_DIR = "TB_Chest_Radiography_Database"

OUTPUT_DIR = "TB_Chest_Radiography_Database_split"

IMG_SIZE = (224, 224)

BATCH_SIZE = 8

SEED = 27

SPLIT_RATIO = [0.8, 0.10, 0.10]  # train, val, test


# --- GPU Setup ---

def setup_gpu():

    gpus = tf.config.experimental.list_physical_devices('GPU')

    if gpus:

        for gpu in gpus:

            tf.config.experimental.set_memory_growth(gpu, True)


# --- Split Dataset (Run Once) ---

def split_dataset():

    if os.path.exists(OUTPUT_DIR):

        print("Dataset already split.")
```

```
        return

    print("Splitting dataset into train/val/test...")
    os.makedirs(OUTPUT_DIR, exist_ok=True)
    for cls in ["Normal", "Tuberculosis"]:
        files = os.listdir(os.path.join(ORIGINAL_DIR, cls))
        random.seed(SEED)
        random.shuffle(files)

        total = len(files)
        train_end = int(total * SPLIT_RATIO[0])
        val_end = train_end + int(total * SPLIT_RATIO[1])

        split_files = {
            "train": files[:train_end],
            "val": files[train_end:val_end],
            "test": files[val_end:]
        }

        for split in split_files:
            split_dir = os.path.join(OUTPUT_DIR, split, cls)
            os.makedirs(split_dir, exist_ok=True)
            for file in tqdm(split_files[split], desc=f"{cls} - {split}"):
                shutil.copy2(os.path.join(ORIGINAL_DIR, cls, file), os.path.join(split_dir, file))

# --- Main Pipeline ---
```

```python
def main():
    setup_gpu() # Setup GPU memory growth
    split_dataset()


if __name__ == "__main__":
    main()
```

Script for Model Evaluation and Comparison (model_comp)

```
import os
import gc
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc, precision_recall_curve, f1_score, accuracy_score
from keras.callbacks import EarlyStopping, ModelCheckpoint
import pandas as pd
import time
from tensorflow.keras.metrics import AUC # type: ignore
from matplotlib.cm import get_cmap



IMG_SIZE = (224, 224)
BATCH_SIZE = 8
EPOCHS = 18
SEED = 27
FOLDS = 5

MODELS = [
    "EfficientNetB0", "MobileNetV2", "ResNet50",
```

```
    "DenseNet121", "InceptionV3", "Xception"
]

CLASS_NAMES = ["Normal", "Tuberculosis"]
SPLITS = ["train", "val", "test"]
BASE_DIR = "TB_Chest_Radiography_Database_split"

all_fpr = {}
all_tpr = {}
all_precision = {}
all_recall = {}



# --- GPU Setup ---
def setup_gpu():
    gpus = tf.config.experimental.list_physical_devices('GPU')
    if gpus:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)


def load_data_as_numpy():
    file_paths = []
    labels = []
    for label, cls in enumerate(CLASS_NAMES):
```

```
        for split in SPLITS:
            dir_path = os.path.join(BASE_DIR, split, cls)
            if os.path.exists(dir_path):
                for fname in os.listdir(dir_path):
                    if fname.lower().endswith((".jpg", ".jpeg", ".png")):
                        file_paths.append(os.path.join(dir_path, fname))
                        labels.append(label)
    return np.array(file_paths), np.array(labels)


def get_model_fn(name):
    models = {
        "EfficientNetB0": tf.keras.applications.EfficientNetB0,
        "MobileNetV2": tf.keras.applications.MobileNetV2,
        "ResNet50": tf.keras.applications.ResNet50,
        "DenseNet121": tf.keras.applications.DenseNet121,
        "InceptionV3": tf.keras.applications.InceptionV3,
        "Xception": tf.keras.applications.Xception
    }
    preprocess = {
        "EfficientNetB0": tf.keras.applications.efficientnet.preprocess_input,
        "MobileNetV2": tf.keras.applications.mobilenet_v2.preprocess_input,
        "ResNet50": tf.keras.applications.resnet.preprocess_input,
        "DenseNet121": tf.keras.applications.densenet.preprocess_input,
        "InceptionV3": tf.keras.applications.inception_v3.preprocess_input,
        "Xception": tf.keras.applications.xception.preprocess_input
```

```python
    }
    return models[name], preprocess[name]


def build_model(model_fn):
    base = model_fn(input_shape=IMG_SIZE + (3,), include_top=False, weights="imagenet")
    base.trainable = False
    x = tf.keras.layers.GlobalAveragePooling2D()(base.output)
    x = tf.keras.layers.Dropout(0.3)(x)
    output = tf.keras.layers.Dense(1, activation="sigmoid")(x)
    return tf.keras.Model(base.input, output)


def create_dataset(paths, labels, preprocess_fn, shuffle=False):
    def _load_image(path, label):
        image = tf.io.read_file(path)
        image = tf.image.decode_image(image, channels=3)
        image.set_shape([None, None, 3])
        image = tf.image.resize(image, IMG_SIZE)
        image = tf.cast(image, tf.float32)
        image = preprocess_fn(image)
        return image, label

    ds = tf.data.Dataset.from_tensor_slices((paths, labels))
    if shuffle:
        ds = ds.shuffle(buffer_size=len(paths), seed=SEED)
```

```
    ds = ds.map(_load_image, num_parallel_calls=tf.data.AUTOTUNE)
    return ds.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)


def count_class_distribution(ds, split_name):
    count_0, count_1 = 0, 0
    for _, labels in ds:
        count_0 += tf.math.count_nonzero(labels == 0).numpy()
        count_1 += tf.math.count_nonzero(labels == 1).numpy()
    print(f"{split_name} split class counts: Normal={count_0}, TB={count_1}")


def plot_class_distribution(y, title, save_path):
    unique, counts = np.unique(y, return_counts=True)
    class_counts = dict(zip(CLASS_NAMES, counts))

    plt.figure(figsize=(6, 4))
    sns.barplot(x=list(class_counts.keys()), y=list(class_counts.values()), palette="pastel")
    plt.title(title)
    plt.ylabel("Number of Samples")
    plt.tight_layout()
    plt.savefig(save_path)
    plt.close()

    plt.figure(figsize=(5, 5))
```

```
    plt.pie(list(class_counts.values()), labels=list(class_counts.keys()), autopct='%1.1f%%',
startangle=140, colors=sns.color_palette("pastel"))
    plt.title(title + " (Pie Chart)")
    plt.tight_layout()
    plt.savefig(save_path.replace(".png", "_pie.png"))
    plt.close()



def train_on_fold(X, y, model_name):
    model_fn, preprocess_fn = get_model_fn(model_name)
    skf = StratifiedKFold(n_splits=FOLDS, shuffle=True, random_state=SEED)
    metrics = []
    model_times = []
    all_y_true, all_y_pred = [], []
    model_conf_matrices = []

    for fold, (train_idx, test_idx) in enumerate(skf.split(X, y)):
        print(f"\n 🧪 Fold {fold+1}/{FOLDS} - {model_name}")

        plot_class_distribution(y[train_idx], f"{model_name} Fold {fold+1} - Train Class
Distribution", f"{model_name}_fold{fold+1}_train_distribution.png")
        plot_class_distribution(y[test_idx], f"{model_name} Fold {fold+1} - Test Class
Distribution", f"{model_name}_fold{fold+1}_test_distribution.png")

        X_train, y_train = X[train_idx], y[train_idx]
        X_test, y_test = X[test_idx], y[test_idx]
```

```
train_ds = create_dataset(X_train, y_train, preprocess_fn, shuffle=True)
test_ds = create_dataset(X_test, y_test, preprocess_fn, shuffle=False)

count_class_distribution(train_ds, "Training")
count_class_distribution(test_ds, "Testing")

model = build_model(model_fn)
model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy",
AUC(name="auc")])

checkpoint_path = f"model_{model_name}_fold{fold+1}.h5"
checkpoint = ModelCheckpoint(checkpoint_path, monitor='val_auc',
save_best_only=True, mode='max', verbose=1)
early_stop = EarlyStopping(patience=3, restore_best_weights=True, monitor='val_loss',
mode='min', verbose=1)

start_time = time.time()
history = model.fit(train_ds, validation_data=test_ds, epochs=EPOCHS,
callbacks=[early_stop, checkpoint], verbose=1)
elapsed_time = time.time() - start_time
model_times.append(elapsed_time)

y_true, y_probs = [], []
# 🔁 Predict in batch
y_probs = model.predict(test_ds)
y_preds = (y_probs.flatten() > 0.5).astype(int)
```

```
# 🔁 Collect true labels
y_true = np.concatenate([label.numpy() for _, label in test_ds], axis=0)



all_y_true.extend(y_true)
all_y_pred.extend(y_preds)
acc = accuracy_score(y_true, y_preds)
f1 = f1_score(y_true, y_preds)
fpr, tpr, _ = roc_curve(y_true, y_probs)
roc_auc = auc(fpr, tpr)
precision, recall, _ = precision_recall_curve(y_true, y_probs)
pr_auc = auc(recall, precision)

all_fpr.setdefault(model_name, []).append(fpr)
all_tpr.setdefault(model_name, []).append(tpr)
all_precision.setdefault(model_name, []).append(precision)
all_recall.setdefault(model_name, []).append(recall)

metrics.append({
  "fold": fold+1,
  "accuracy": acc,
  "f1": f1,
  "roc_auc": roc_auc,
  "pr_auc": pr_auc,
  "training_time": elapsed_time
})
```

```
    cm = confusion_matrix(y_true, y_preds)
    model_conf_matrices.append(cm)
    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
    plt.title(f"Confusion Matrix - {model_name} Fold {fold+1}")
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.tight_layout()
    plt.savefig(f"cm_{model_name}_fold{fold+1}.png")
    plt.close()



    report = classification_report(y_true, y_preds, target_names=CLASS_NAMES)
    with open(f"classification_report_{model_name}_fold{fold+1}.txt", "w") as f:
        f.write(report)



    tf.keras.backend.clear_session()  # Clear TensorFlow session first
    del model, history, test_ds, train_ds, X_train, X_test, y_train, y_test
    gc.collect()  # Trigger garbage collection to free memory
    print(f"Memory cleared for {model_name} after fold {fold+1}...")

# Compute global statistics across folds
```

```
    global_metrics = {
        "accuracy_mean": np.mean([m["accuracy"] for m in metrics]),
        "accuracy_std": np.std([m["accuracy"] for m in metrics]),
        "f1_mean": np.mean([m["f1"] for m in metrics]),
        "f1_std": np.std([m["f1"] for m in metrics]),
        "roc_auc_mean": np.mean([m["roc_auc"] for m in metrics]),
        "roc_auc_std": np.std([m["roc_auc"] for m in metrics]),
        "pr_auc_mean": np.mean([m["pr_auc"] for m in metrics]),
        "pr_auc_std": np.std([m["pr_auc"] for m in metrics]),
        "training_time_mean": np.mean(model_times),
        "training_time_std": np.std(model_times)
    }

    return metrics,model_conf_matrices,global_metrics

def plot_avg_confusion_matrices(models_conf_matrices):
    for model_name, cm_list in models_conf_matrices.items():
        avg_cm = np.mean(cm_list, axis=0)
        plt.figure(figsize=(5, 4))
        sns.heatmap(avg_cm, annot=True, fmt=".1f", cmap="Blues")
        plt.title(f"Avg Confusion Matrix - {model_name}")
        plt.xlabel("Predicted")
        plt.ylabel("True")
        plt.tight_layout()
        plt.savefig(f"avg_cm_{model_name}.png")
        plt.close()
```

```
def plot_mean_roc_pr_curves():
   color_map = get_cmap('tab10')  # 10 visually distinct colors
   model_colors = {model: color_map(i) for i, model in enumerate(MODELS)}

   for curve_type, all_curves, xlabel, ylabel, title, filename in [
      ("ROC", (all_fpr, all_tpr), "False Positive Rate", "True Positive Rate", "Mean ROC Curve",
"mean_roc_comparison.png"),
      ("PR", (all_recall, all_precision), "Recall", "Precision", "Mean Precision-Recall Curve",
"mean_pr_comparison.png")
   ]:
      plt.figure(figsize=(10, 7))

      for idx, model_name in enumerate(MODELS):
         xs = np.linspace(0, 1, 100)
         ys_interp = []
         if model_name in all_curves[0]:
            for fold_x, fold_y in zip(all_curves[0][model_name], all_curves[1][model_name]):
               interp_y = np.interp(xs, fold_x, fold_y)
               ys_interp.append(interp_y)

            if ys_interp:
               mean_y = np.mean(ys_interp, axis=0)
               std_y = np.std(ys_interp, axis=0)
```

```
            plt.plot(xs, mean_y, label=model_name, color=model_colors[model_name])
            plt.fill_between(xs, mean_y - std_y, mean_y + std_y, alpha=0.2,
color=model_colors[model_name])

    plt.xlabel(xlabel, fontsize=12)
    plt.ylabel(ylabel, fontsize=12)
    plt.title(title, fontsize=14)
    plt.legend(loc='lower right' if curve_type == "ROC" else 'lower left', fontsize=10)
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.savefig(filename, dpi=300)
    plt.close()


def plot_roc_pr_auc_comparison(models_metrics):
  roc_auc_means = []
  pr_auc_means = []
  models = []

  for model_name, metrics in models_metrics.items():
    roc_auc_scores = [m["roc_auc"] for m in metrics]
    pr_auc_scores = [m["pr_auc"] for m in metrics]

    roc_auc_means.append(np.mean(roc_auc_scores))
    pr_auc_means.append(np.mean(pr_auc_scores))
    models.append(model_name)
```

```
    plt.figure(figsize=(10, 6))
    plt.plot(models, roc_auc_means, label="ROC AUC", marker='o')
    plt.plot(models, pr_auc_means, label="PR AUC", marker='o')
    plt.title("ROC AUC and PR AUC Comparison Across Models")
    plt.xlabel("Model")
    plt.ylabel("AUC Score")
    plt.legend()
    plt.tight_layout()
    plt.savefig("roc_pr_auc_comparison.png")
    plt.close()


def plot_global_metrics_bar(global_metrics_dict):
    metrics_to_plot = ["accuracy", "f1", "roc_auc", "pr_auc", "training_time"]
    means = {m: [] for m in metrics_to_plot}
    stds = {m: [] for m in metrics_to_plot}
    model_names = list(global_metrics_dict.keys())

    for m in metrics_to_plot:
        for model in model_names:
            means[m].append(global_metrics_dict[model][f"{m}_mean"])
            stds[m].append(global_metrics_dict[model][f"{m}_std"])

    for m in metrics_to_plot:
        plt.figure(figsize=(10, 6))
```

```
    plt.bar(model_names, means[m], yerr=stds[m], capsize=5, color="skyblue")
    plt.ylabel(f"{m.replace('_', ' ').title()}")
    plt.title(f"Global {m.replace('_', ' ').title()} Comparison Across Models")
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.savefig(f"global_{m}_comparison.png")
    plt.close()



def plot_training_time(models_metrics):
  for model_name, metrics in models_metrics.items():
    training_times = [m["training_time"] for m in metrics]

    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(metrics) + 1), training_times, label="Training Time (seconds)",
marker='o')
    plt.title(f"{model_name} - Training Time per Fold")
    plt.xlabel("Fold")
    plt.ylabel("Time (seconds)")
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"{model_name}_training_time.png")
    plt.close()
```

```
def plot_bar_with_error(models_metrics, metric_key, title, ylabel):
    names = []
    means = []
    stds = []

    for model, metrics in models_metrics.items():
        vals = [m[metric_key] for m in metrics]
        names.append(model)
        means.append(np.mean(vals))
        stds.append(np.std(vals))

    plt.figure(figsize=(10, 6))
    plt.bar(names, means, yerr=stds, capsize=5, color='skyblue')
    plt.title(title)
    plt.ylabel(ylabel)
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.savefig(f"{metric_key}_comparison.png")
    plt.close()


def plot_metrics_boxplots(all_model_metrics):
    # Include train_time
    metric_names = ["accuracy", "f1", "roc_auc", "pr_auc", "training_time"]  # Corrected to use "training_time"
```

```
    for metric in metric_names:
        data = []
        labels = []
        for model_name, metrics in all_model_metrics.items():
            values = [m[metric] for m in metrics]  # Make sure to extract training time
(elapsed_time) here
            data.append(values)
            labels.append(model_name)

        plt.figure(figsize=(10, 6))
        sns.boxplot(data=data)
        plt.xticks(ticks=range(len(labels)), labels=labels, rotation=45)
        plt.ylabel("Training Time (s)" if metric == "training_time" else metric.replace("_", "
").title())
        plt.title(f"{'Training Time' if metric == 'training_time' else metric.replace('_', ' ').title()}
Comparison Across Models")
        plt.tight_layout()
        plt.savefig(f"boxplot_{metric}.png")
        plt.close()



def compare_models_summary(
    all_global_metrics,
    save_csv="model_summary.csv",
    save_heatmap="model_metrics_heatmap.png",
```

```
    save_roc_bar="roc_auc_comparison_bar.png",
    save_roc_pr_bar="roc_vs_pr_auc_barplot.png",
    save_training_time_bar="training_time_barplot.png"
):
    """
    Compare global metrics across models using ROC-AUC (primary) and PR-AUC (secondary).
    Improved version: Fixes heatmap highlight, separates training time, adds PR-ROC delta.
    """
    # Create DataFrame
    df = pd.DataFrame.from_dict(all_global_metrics, orient="index")

    # Add PR-ROC delta column
    df["pr_roc_delta"] = df["pr_auc_mean"] - df["roc_auc_mean"]

    # Sort by roc_auc_mean, then pr_auc_mean
    df_sorted = df.sort_values(by=["roc_auc_mean", "pr_auc_mean"], ascending=False)
    best_model = df_sorted.index[0]

    print(f"🏆 Best model based on ROC-AUC + PR-AUC fallback: {best_model}\n")
    print(df_sorted.loc[[best_model]])

    # Save CSV
    df_rounded = df.round(4)
    df_rounded.to_csv(save_csv)
```

```
# -------------------- HEATMAP (Performance Metrics) --------------------
mean_cols = [col for col in df.columns if "mean" in col and col != "training_time_mean"]
heatmap_data = df_sorted[mean_cols]

plt.figure(figsize=(10, 6))
ax = sns.heatmap(heatmap_data, annot=True, fmt=".3f", cmap="viridis",
cbar_kws={"label": "Score"})

# Highlight best model row (now correctly aligned with sorted data)
ax.add_patch(plt.Rectangle((0, 0), len(heatmap_data.columns), 1, color="red",
alpha=0.3))

plt.title("Model Performance Comparison (Mean Metrics)")
plt.tight_layout()
plt.savefig(save_heatmap)
plt.close()

# -------------------- ROC-AUC BAR PLOT --------------------
plt.figure(figsize=(8, 5))
df_sorted["roc_auc_mean"].plot(kind="bar", color="skyblue", edgecolor="black")
plt.title("Model Comparison: ROC-AUC")
plt.ylabel("ROC-AUC Score")
plt.xticks(rotation=45)
plt.tight_layout()
plt.savefig(save_roc_bar)
plt.close()
```

```python
    # -------------------- ROC-AUC vs PR-AUC BAR PLOT --------------------
    df_sorted[["roc_auc_mean", "pr_auc_mean"]].plot(kind="bar", figsize=(10, 6))
    plt.title("Model Comparison: ROC-AUC vs PR-AUC")
    plt.ylabel("Score")
    plt.xticks(rotation=45)
    plt.legend(loc="lower right")
    plt.tight_layout()
    plt.savefig(save_roc_pr_bar)
    plt.close()

    # -------------------- TRAINING TIME BAR PLOT --------------------
    plt.figure(figsize=(8, 5))
    df_sorted["training_time_mean"].plot(kind="bar", color="orange", edgecolor="black")
    plt.title("Model Comparison: Training Time (Mean)")
    plt.ylabel("Time (seconds or units)")
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.savefig(save_training_time_bar)
    plt.close()

    return best_model


def save_metrics_csv(models_metrics):
    rows = []
```

```
    for model, metrics in models_metrics.items():
        for m in metrics:
            row = {"model": model, **m}
            rows.append(row)
    pd.DataFrame(rows).to_csv("model_comparison_cv.csv", index=False)



def main():

    setup_gpu()  # Setup GPU memory growth
    X, y = load_data_as_numpy()
    plot_class_distribution(y, title="Overall Dataset Class Distribution",
save_path="overall_class_distribution.png")

    models_metrics = {}
    models_cms = {}
    all_global_metrics = {}

    for model_name in MODELS:
        metrics, cms, global_metrics = train_on_fold(X, y, model_name)
        models_metrics[model_name] = metrics
        models_cms[model_name] = cms
        all_global_metrics[model_name] = global_metrics

    save_metrics_csv(models_metrics)
```

```
    # Plot error bar charts for various metrics
    plot_bar_with_error(models_metrics, "accuracy", "Test Accuracy Comparison",
"Accuracy")
    plot_bar_with_error(models_metrics, "f1", "F1 Score Comparison", "F1 Score")
    plot_bar_with_error(models_metrics, "roc_auc", "ROC AUC Comparison", "ROC AUC")
    plot_bar_with_error(models_metrics, "pr_auc", "PR AUC Comparison", "PR AUC")
    plot_bar_with_error(models_metrics, "training_time", "Training Time Comparison", "Time
(s)")

    # Plot global metrics comparison
    plot_global_metrics_bar(all_global_metrics)

    # Plot ROC and PR curves comparison
    plot_roc_pr_auc_comparison(models_metrics)

    # Plot training time comparison
    plot_training_time(models_metrics)

    # Plot avg confusion matrices for each model
    plot_avg_confusion_matrices(models_cms)

    # Plot mean ROC and PR curves
    plot_mean_roc_pr_curves()

    # Call the new boxplot plotting function for all metrics
    plot_metrics_boxplots(models_metrics)
```

```
    # Compare models and get the best model
    best_model = compare_models_summary(all_global_metrics)
    # Output the best model
    print(f"The best model is: {best_model}")

if __name__ == "__main__":
    main()
```

Script for Hyperparameter Tuning (hypertune_final.py).

```
import os
import pickle
import tensorflow as tf
from tensorflow import keras
from keras import layers
from keras.applications import ResNet50, resnet
from keras.models import Model
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras_tuner.tuners import Hyperband
from keras_tuner import Objective

import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Set GPU memory growth
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    for gpu in gpus:
        tf.config.experimental.set_memory_growth(gpu, True)

# Configs
IMG_SIZE = (224, 224)
BATCH_SIZE = 8  # Increased batch size for stability
SEED = 27
```

```
OUTPUT_DIR = "TB_Chest_Radiography_Database_split"

PLOTS_DIR = "plots/tuning"

os.makedirs(PLOTS_DIR, exist_ok=True)


# Load datasets

def load_datasets():

  train_ds = tf.keras.preprocessing.image_dataset_from_directory(

    os.path.join(OUTPUT_DIR, "train"), seed=SEED, image_size=IMG_SIZE,

    batch_size=BATCH_SIZE, label_mode="binary")

  val_ds = tf.keras.preprocessing.image_dataset_from_directory(

    os.path.join(OUTPUT_DIR, "val"), seed=SEED, image_size=IMG_SIZE,

    batch_size=BATCH_SIZE, label_mode="binary")

  return train_ds, val_ds


train_ds, val_ds = load_datasets()


# Preprocess

train_ds = train_ds.map(lambda x, y: (resnet.preprocess_input(x),
y)).prefetch(tf.data.AUTOTUNE)

val_ds = val_ds.map(lambda x, y: (resnet.preprocess_input(x),
y)).prefetch(tf.data.AUTOTUNE)


# Build model with tunable hyperparameters

def build_model(hp):

  base_model = ResNet50(include_top=False, input_shape=IMG_SIZE + (3,),
weights="imagenet")

  base_model.trainable = False
```

```
    x = layers.GlobalAveragePooling2D()(base_model.output)
    x = layers.Dropout(hp.Float('dropout', min_value=0.1, max_value=0.6, step=0.1))(x)  #
Expanded range
    x = layers.Dense(hp.Int('dense_units', min_value=128, max_value=1024, step=128),
activation='relu')(x)  # Expanded range
    x = layers.Dropout(hp.Float('dropout_2', min_value=0.1, max_value=0.6, step=0.1))(x)  #
Expanded range
    output = layers.Dense(1, activation='sigmoid')(x)

    model = Model(inputs=base_model.input, outputs=output)

    model.compile(
        optimizer=keras.optimizers.Adam(
            hp.Choice('learning_rate', [0.005, 0.003, 0.001, 0.0005])  # Adjusted learning rate
values
        ),
        loss='binary_crossentropy',
        metrics=['accuracy', 'AUC', 'Precision', 'Recall']  # Added more metrics
    )

    return model

# Hyperband tuner for optimizing val_auc
tuner = Hyperband(
    build_model,
    objective=Objective("val_auc", direction="max"),
```

```
    max_epochs=30,
    factor=4,
    directory='resnet50_tuning_v2',
    project_name='tb_classification_v2'
)

# Callbacks for tuning
stop_early = EarlyStopping(patience=4,  monitor='val_loss', mode='min', verbose=1)
model_checkpoint_cb = ModelCheckpoint(
    os.path.join(PLOTS_DIR, "best_model_ResNet50.h5"),
    save_best_only=True,
    monitor="val_auc",
    mode="max",
    verbose=1
)

# Search best hyperparameters
tuner.search(train_ds, validation_data=val_ds, epochs=30, callbacks=[stop_early,
model_checkpoint_cb])

# Retrieve best model & hyperparameters
best_model = tuner.get_best_models(num_models=1)[0]
best_hps = tuner.get_best_hyperparameters(1)[0]
# Save best hyperparameters
with open("best_hyperparameters.pkl", "wb") as f:
    pickle.dump(best_hps.values, f)
```

```
print("\nBest Hyperparameters:")
for param in ['dropout', 'dropout_2', 'dense_units', 'learning_rate']:
    print(f"{param}: {best_hps.get(param)}")

# Save tuner-best model
best_model.save('best_resnet50_tuned.h5')
print(f"Saved tuned model to 'best_resnet50_tuned.h5'")

# Plot & save functions
def save_plot(fig, name):
    fig.savefig(os.path.join(PLOTS_DIR, name), bbox_inches='tight')


# Visualize hyperparameter tuning results
def visualize_tuning_results(tuner):
    trials = tuner.oracle.get_best_trials(num_trials=80)  # Get top 80 trials

    if not trials:
        print("No trials found in the tuner.")
        return

    # Extract data
    trial_ids = [trial.trial_id for trial in trials]
    val_accuracies = [trial.metrics.get_last_value('val_accuracy') for trial in trials]
    val_losses = [trial.metrics.get_last_value('val_loss') for trial in trials]
```

```
val_aucs = [trial.metrics.get_last_value('val_auc') for trial in trials]
dropouts = [trial.hyperparameters.get('dropout') for trial in trials]
dropouts_2 = [trial.hyperparameters.get('dropout_2') for trial in trials]
dense_units = [trial.hyperparameters.get('dense_units') for trial in trials]
learning_rates = [trial.hyperparameters.get('learning_rate') for trial in trials]

# Plot: Validation Accuracy per Trial (Line)
plt.figure(figsize=(10, 6))
plt.plot(val_accuracies, marker='o', linestyle='-')
plt.title('Validation Accuracy per Trial')
plt.xlabel('Trial Index')
plt.ylabel('Validation Accuracy')
plt.grid(True)
save_plot(plt, 'val_accuracy_per_trial.png')
plt.close()

# Plot: Validation Loss per Trial (Line)
plt.figure(figsize=(10, 6))
plt.plot(val_losses, marker='o', linestyle='-', color='red')
plt.title('Validation Loss per Trial')
plt.xlabel('Trial Index')
plt.ylabel('Validation Loss')
plt.grid(True)
save_plot(plt, 'val_loss_per_trial.png')
plt.close()
```

```
# Plot: Validation AUC per Trial (Line)
plt.figure(figsize=(10, 6))
plt.plot(val_aucs, marker='o', linestyle='-', color='green')
plt.title('Validation AUC per Trial')
plt.xlabel('Trial Index')
plt.ylabel('Validation AUC')
plt.grid(True)
save_plot(plt, 'val_auc_per_trial.png')
plt.close()


# Scatter Plot: Validation Loss vs Accuracy
plt.figure(figsize=(8, 6))
plt.scatter(val_losses, val_accuracies, c='teal', edgecolors='black')
plt.xlabel('Validation Loss')
plt.ylabel('Validation Accuracy')
plt.title('Validation Loss vs Accuracy')
plt.grid(True)
plt.tight_layout()
plt.savefig(os.path.join(PLOTS_DIR, "val_loss_vs_val_acc.png"))
plt.close()


# Scatter Plot: Validation Loss vs AUC
val_aucs = [trial.metrics.get_last_value('val_auc') for trial in trials]
plt.figure(figsize=(8, 6))
```

```
plt.scatter(val_losses, val_aucs, c='darkorange', edgecolors='black')
plt.xlabel('Validation Loss')
plt.ylabel('Validation AUC')
plt.title('Validation Loss vs AUC')
plt.grid(True)
plt.tight_layout()
plt.savefig(os.path.join(PLOTS_DIR, "val_loss_vs_val_auc.png"))
plt.close()

# Plot: Hyperparameter Grid Search (Heatmap of Hyperparameters vs AUC)
hyperparams_df = pd.DataFrame({
    'Dropout': dropouts,
    'Dropout_2': dropouts_2,
    'Dense Units': dense_units,
    'Learning Rate': learning_rates,
    'AUC': val_aucs
})
pivot_df = hyperparams_df.pivot_table(
    index='Dense Units', columns='Learning Rate', values='AUC', aggfunc='mean'
)

plt.figure(figsize=(12, 8))
sns.heatmap(pivot_df, annot=True, cmap='viridis', fmt=".3f", linewidths=0.5)
plt.title('Hyperparameter Grid Search (AUC vs Dense Units & Learning Rate)')
plt.xlabel('Learning Rate')
plt.ylabel('Dense Units')
```

```
    save_plot(plt, 'hyperparameter_grid_search_heatmap.png')
    plt.close()

    # Best Trial Summary
    best_trial = trials[0]  # The best trial is now the first one in the list
    print("\nBest Trial Hyperparameters:")
    for param, value in best_trial.hyperparameters.values.items():
        print(f"{param}: {value}")
    print(f"Best Validation Accuracy: {best_trial.metrics.get_best_value('val_accuracy')}")
    print(f"Best Validation Loss: {best_trial.metrics.get_best_value('val_loss')}")
    print(f"Best Validation AUC: {best_trial.metrics.get_best_value('val_auc')}")

# Visualize tuning results after finding the best model
visualize_tuning_results(tuner)
```

Script for Final Model Training and Evaluation(final_training.py)

```
import os

import pickle

import tensorflow as tf

from keras import layers

from keras.applications import ResNet50, resnet

from keras.models import Model

from keras.callbacks import EarlyStopping, ModelCheckpoint

import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import confusion_matrix, roc_curve, auc, precision_recall_curve,
f1_score

import math # Import math for splitting dataset


# Set GPU memory growth

gpus = tf.config.experimental.list_physical_devices('GPU')

if gpus:

  for gpu in gpus:

    tf.config.experimental.set_memory_growth(gpu, True)


# Configs

IMG_SIZE = (224, 224)

BATCH_SIZE = 8

SEED = 27 # Use the same seed for reproducibility

OUTPUT_DIR = "TB_Chest_Radiography_Database_split"

PLOTS_DIR = "plots/final_training" # Separate plots for final training
```

```
os.makedirs(PLOTS_DIR, exist_ok=True)


# Save plot utility
def save_plot(fig, name):
  fig.savefig(os.path.join(PLOTS_DIR, name), bbox_inches='tight')
  plt.close(fig) # Close the figure after saving


# Load datasets
# We need train_ds, val_ds, and test_ds separately first to combine train+val
# and keep test completely separate.
def load_datasets():
  train_ds = tf.keras.preprocessing.image_dataset_from_directory(
    os.path.join(OUTPUT_DIR, "train"),
    seed=SEED,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    label_mode="binary"
  )
  val_ds = tf.keras.preprocessing.image_dataset_from_directory(
    os.path.join(OUTPUT_DIR, "val"),
    seed=SEED,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    label_mode="binary"
  )
  test_ds = tf.keras.preprocessing.image_dataset_from_directory(
```

```
        os.path.join(OUTPUT_DIR, "test"),
        seed=SEED,
        image_size=IMG_SIZE,
        batch_size=BATCH_SIZE,
        label_mode="binary"
    )
    return train_ds, val_ds, test_ds

# Load datasets (they are initially batched)
train_ds_batched, val_ds_batched, test_ds_batched_initial = load_datasets() # Use temporary names for clarity

# --- Data Preparation for Final Training ---
# Unbatch initial datasets first to work with individual samples
train_ds_unbatched = train_ds_batched.unbatch()
val_ds_unbatched = val_ds_batched.unbatch()
test_ds_unbatched = test_ds_batched_initial.unbatch() # Also unbatch test now for consistent processing

# Combine original train and validation samples (now unbatched)
combined_ds_unbatched = train_ds_unbatched.concatenate(val_ds_unbatched)

# Calculate the size of the combined dataset (now unbatched)
# This loop counts individual samples
print("Calculating combined dataset size...")
combined_ds_unbatched_size = 0
for _ in combined_ds_unbatched:
```

```
    combined_ds_unbatched_size += 1
print(f"Combined Train + Val Dataset Size: {combined_ds_unbatched_size} samples")

# Define the split ratios for the final training (based on unbatched samples)
FINAL_TRAIN_RATIO = 0.8
FINAL_VAL_RATIO = 1.0 - FINAL_TRAIN_RATIO # Should be 0.2

final_train_size_unbatched = math.floor(FINAL_TRAIN_RATIO * combined_ds_unbatched_size)
final_val_size_unbatched = combined_ds_unbatched_size - final_train_size_unbatched

print(f"Final Training Set Size (from combined): {final_train_size_unbatched} samples")
print(f"Final Validation Set Size (from combined): {final_val_size_unbatched} samples")
# Alternative way to get actual test set size for printing:
print("Calculating test dataset size for printing...")
test_ds_actual_size = 0
# Iterate over the UNBATCHED test set to count samples
for _ in test_ds_unbatched:
    test_ds_actual_size += 1
print(f"Test Set Size (original unbatched): {test_ds_actual_size} samples")

# Shuffle the combined unbatched dataset
combined_ds_unbatched = combined_ds_unbatched.shuffle(buffer_size=combined_ds_unbatched_size, seed=SEED)

# Split the combined unbatched dataset into final training and final validation sets (still unbatched)
```

```
final_train_ds_unbatched = combined_ds_unbatched.take(final_train_size_unbatched)

final_val_ds_unbatched = combined_ds_unbatched.skip(final_train_size_unbatched)


# --- Now Batch, Cache, and Prefetch the Split Datasets ---

# Batch the individual samples into desired batch sizes

final_train_ds =
final_train_ds_unbatched.batch(BATCH_SIZE).cache().prefetch(tf.data.AUTOTUNE)

final_val_ds =
final_val_ds_unbatched.batch(BATCH_SIZE).cache().prefetch(tf.data.AUTOTUNE)

# Batch the test set consistently as well

test_ds = test_ds_unbatched.batch(BATCH_SIZE).cache().prefetch(tf.data.AUTOTUNE) #
Use 'test_ds' for final batched dataset


# --- Preprocessing (ResNet specific) ---

# Apply preprocess_input AFTER batching but before prefetching


def preprocess_batched_dataset(dataset):

    # Preprocessing is applied to batches returned by .batch()

    dataset = dataset.map(lambda x, y: (resnet.preprocess_input(x), y))

    return dataset.prefetch(tf.data.AUTOTUNE) # Prefetch after transformations


# Apply preprocessing to the newly batched datasets

final_train_ds = preprocess_batched_dataset(final_train_ds)

final_val_ds = preprocess_batched_dataset(final_val_ds)

test_ds = preprocess_batched_dataset(test_ds) # Apply to the final batched test_ds


# --- End of Corrected Data Preparation Section ---
```

```
# --- Load Best Hyperparameters ---
try:
   with open("best_hyperparameters.pkl", "rb") as f:
      best_hps_values = pickle.load(f)
   print("\nLoaded Best Hyperparameters:")
   for param in ['dropout', 'dropout_2', 'dense_units', 'learning_rate']:
      if param in best_hps_values:
         print(f"{param}: {best_hps_values.get(param)}")
      else:
         print(f"{param}: Not found in loaded hyperparameters")

except FileNotFoundError:
   print("Error: best_hyperparameters.pkl not found.")
   print("Please run the hyperparameter tuning script (script 1) first to generate this file.")
   exit()

# --- Build Final Model (consistent with tuning architecture) ---
def build_final_model(hparams):
   # Ensure the base model remains frozen, as it was during tuning
   base_model = ResNet50(include_top=False, input_shape=IMG_SIZE + (3,),
weights="imagenet")
   base_model.trainable = False # Keep base model frozen

   x = layers.GlobalAveragePooling2D()(base_model.output)
   # Use dropout rates from tuning
```

```
    x = layers.Dropout(hparams['dropout'])(x)
    # Use dense units from tuning, NO L2 regularization to match tuning setup
    x = layers.Dense(
        hparams['dense_units'],
        activation='relu'
        # Removed: kernel_regularizer=regularizers.l2(0.001)
    )(x)
    # Use second dropout rate from tuning
    x = layers.Dropout(hparams['dropout_2'])(x)
    output = layers.Dense(1, activation='sigmoid')(x)

    model = Model(inputs=base_model.input, outputs=output)
    return model

# Build the final model
final_model = build_final_model(best_hps_values)

# --- Compile Final Model ---
# Use the exact learning rate found by the tuner
final_model.compile(
    optimizer=tf.keras.optimizers.Adam(best_hps_values['learning_rate']), # Use tuner's LR directly
    loss='binary_crossentropy',
    metrics=['accuracy', 'AUC', 'Precision', 'Recall']
)
```

```
# --- Callbacks for Final Training ---
early_stopping = EarlyStopping(
    patience=8, # Increased patience slightly for training on larger data
    monitor='val_loss', # Monitor the loss on the temporary val set
    mode='min',
    verbose=1,
    restore_best_weights=True # Restore weights from the epoch with best val performance
)

# Save the best model weights during this final training run based on validation AUC
model_checkpoint_cb = ModelCheckpoint(
    os.path.join(PLOTS_DIR, "final_best_model_ResNet50.weights.h5"), # Save only weights
    save_best_only=True,
    monitor='val_auc', # Monitor val_auc to align with tuning objective
    mode='max', # We want to maximize AUC
    verbose=1
)

# --- Train on the Final Combined Dataset ---
# Train for enough epochs to allow Early Stopping to trigger
print("\nStarting final model training on combined dataset...")
history = final_model.fit(
    final_train_ds,
    validation_data=final_val_ds, # Validate on the temporary split
    epochs=100, # Set a sufficiently large number of epochs, EarlyStopping will stop it
    callbacks=[early_stopping, model_checkpoint_cb]
```

```python
)

print("\nFinal model training finished.")

# --- Plotting Training History ---
print("Generating training history plots...")

# Get history data
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
train_auc_history = history.history['auc'] # Renamed
val_auc_history = history.history['val_auc'] # Renamed
epochs_range = range(len(acc))

# Accuracy Plot
fig = plt.figure(figsize=(8, 6))
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
save_plot(fig, 'training_validation_accuracy.png')
```

```
# Loss Plot
fig = plt.figure(figsize=(8, 6))
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
save_plot(fig, 'training_validation_loss.png')


# AUC Plot
fig = plt.figure(figsize=(8, 6))
plt.plot(epochs_range, train_auc_history, label='Training AUC') # Use the new variable
name
plt.plot(epochs_range, val_auc_history, label='Validation AUC') # Use the new variable
name
plt.legend(loc='lower right')
plt.title('Training and Validation AUC')
plt.xlabel('Epoch')
plt.ylabel('AUC')
save_plot(fig, 'training_validation_auc.png')


# Load the best weights saved by the checkpoint
final_model.load_weights(os.path.join(PLOTS_DIR,
"final_best_model_ResNet50.weights.h5"))
print("\n\nLoaded best weights from final training for evaluation.")
```

```
# --- Evaluate on the Test Set ---

print("\nEvaluating final model on the test set...")

y_true, y_pred_classes, y_score = [], [], []


# Iterate directly over the final correctly batched and preprocessed test_ds

for x_batch, y_batch in test_ds: # test_ds is now the final batched dataset from the corrected pipeline

    y_true.extend(y_batch.numpy())

    preds = final_model.predict(x_batch, verbose=0) # Predict on the batch

    y_score.extend(preds.ravel()) # Store probabilities

    y_pred_classes.extend((preds.ravel() > 0.5).astype(int)) # Store predicted classes


y_true = np.array(y_true)

y_pred_classes = np.array(y_pred_classes)

y_score = np.array(y_score)


# --- Calculate and Plot Metrics ---

print("\nCalculating test set metrics...")


# Test Accuracy

test_acc = np.mean(y_pred_classes == y_true)


# ROC Curve and AUC

fpr, tpr, _ = roc_curve(y_true, y_score)

roc_auc = auc(fpr, tpr)
```

```
# Precision-Recall Curve
precision, recall, _ = precision_recall_curve(y_true, y_score)
# Note: Precision_recall_curve does not return a single score like AUC.
# We can calculate average precision if needed, or just plot the curve.
# Let's calculate F1 score based on the 0.5 threshold
f1 = f1_score(y_true, y_pred_classes)

# Confusion Matrix
cm = confusion_matrix(y_true, y_pred_classes)

print(f"Test Accuracy: {test_acc:.4f}")
print(f"Test AUC: {roc_auc:.4f}")
print(f"Test F1 Score (threshold 0.5): {f1:.4f}")
print("Confusion Matrix:\n", cm)

# Plotting
print("Generating plots...")

# Confusion matrix plot
fig = plt.figure(figsize=(6, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
tick_marks = np.arange(2)
```

```
plt.xticks(tick_marks, ['Normal (0)', 'TB (1)'])
plt.yticks(tick_marks, ['Normal (0)', 'TB (1)'])
# Add text labels
thresh = cm.max() / 2.
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, format(cm[i, j], 'd'),
                 ha="center", va="center",
                 color="white" if cm[i, j] > thresh else "black")
save_plot(fig, 'confusion_matrix.png')

# ROC Curve plot
fig = plt.figure(figsize=(6, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
save_plot(fig, 'roc_curve.png')

# Precision-Recall Curve plot
fig = plt.figure(figsize=(6, 6))
plt.plot(recall, precision, color='teal', lw=2, label='Precision-Recall curve')
```

```
plt.xlabel('Recall')

plt.ylabel('Precision')

plt.title('Precision-Recall Curve')

plt.ylim([0.0, 1.05])

plt.xlim([0.0, 1.0])

plt.legend(loc="lower left")

save_plot(fig, 'pr_curve.png')


# Histogram of probabilities

fig = plt.figure(figsize=(8, 5))

plt.hist(y_score, bins=50, edgecolor='black', alpha=0.7)

plt.title('Prediction Probability Distribution on Test Set')

plt.xlabel('Predicted Probability ( closer to 0 is Normal, closer to 1 is TB)')

plt.ylabel('Frequency')

save_plot(fig, 'probability_histogram.png')


# True vs predicted count (on test set)

fig = plt.figure(figsize=(6, 5))

bar_x = np.arange(2)

counts_true = [np.sum(y_true == 0), np.sum(y_true == 1)]

counts_pred = [np.sum(y_pred_classes == 0), np.sum(y_pred_classes == 1)]

plt.bar(bar_x - 0.1, counts_true, width=0.2, label='True')

plt.bar(bar_x + 0.1, counts_pred, width=0.2, label='Predicted')

plt.xticks(bar_x, ['Normal (0)', 'TB (1)'])

plt.ylabel('Count')

plt.title("True vs Predicted Counts on Test Set")
```

```
plt.legend()
save_plot(fig, 'true_vs_predicted_counts.png')

print(f"\nEvaluation complete. Plots saved to {PLOTS_DIR}/")

# --- Save the final model for deployment ---
final_model_path = "final_model_for_deployment.h5"

final_model.save(final_model_path)
print(f"\nFinal model for deployment saved to '{final_model_path}'")
```

Script for Model Deployment in Streamlit App (app.py)

```
# --- Tuberculosis Detection App ---


import streamlit as st

import tensorflow as tf

import numpy as np

from keras.models import load_model

from keras.utils import load_img, img_to_array


# --- Page Config ---

st.set_page_config(page_title="TB Detection", layout="centered", page_icon="🩻")


# --- Constants ---

MODEL_PATH = "best_resnet50_tuned.h5"

IMG_SIZE = (224, 224)


# --- Load model ---

@st.cache(allow_output_mutation=True)

def load_model():

    return tf.keras.models.load_model(MODEL_PATH)


model = load_model()


# --- Styling ---

st.markdown("""

    <style>
```

```css
html, body, [class*="css"] {
  background-color: #121212;
  color: #E0E0E0;
  font-family: 'Segoe UI', sans-serif;
}
.title {
  font-size: 2.8em;
  font-weight: 600;
  text-align: center;
  margin-bottom: 0.2em;
  color: #FAFAFA;
}
.subtitle {
  font-size: 1.2em;
  text-align: center;
  margin-bottom: 2em;
  color: #B0B0B0;
}
.result-card {
  background-color: #1E1E1E;
  padding: 1.5em;
  border-radius: 10px;
  text-align: center;
  box-shadow: 0 4px 12px rgba(0,0,0,0.4);
  margin-top: 2em;
}
```

```
    .result-label {
        font-size: 1.8em;
        font-weight: bold;
        color: #4FC3F7;
    }
    .positive {
        color: #EF5350;
    }
    .negative {
        color: #66BB6A;
    }
    .confidence {
        margin-top: 0.5em;
        font-size: 1.1em;
        color: #CCCCCC;
    }
    .footer {
        text-align: center;
        font-size: 0.9em;
        color: #888888;
        margin-top: 3em;
    }
    </style>
""", unsafe_allow_html=True)

# --- Title ---
```

```
st.markdown('<div class="title">🩺 Tuberculosis Detection from Chest X-rays</div>',
unsafe_allow_html=True)

st.markdown('<div class="subtitle">Upload a chest X-ray image (JPG or PNG) to check for
Tuberculosis</div>', unsafe_allow_html=True)


# --- Image Preprocessing ---

def preprocess_image(uploaded_file):

    img = load_img(uploaded_file, target_size=IMG_SIZE)

    img = img.convert("RGB")

    img_array = img_to_array(img)

    img_array = tf.keras.applications.resnet50.preprocess_input(img_array)

    return np.expand_dims(img_array, axis=0)


# --- File uploader ---

uploaded_file = st.file_uploader("📤 Upload Chest X-ray", type=["jpg", "jpeg", "png"])


if uploaded_file:

    # Display image using raw bytes (Streamlit supports this)

    st.image(uploaded_file, caption="📷 Uploaded X-ray", use_column_width=True)


    col1, col2 = st.columns([1, 1])


    with col1:

        if st.button("🔎 Analyze X-ray"):

            with st.spinner("Analyzing with deep learning model..."):

                processed_img = preprocess_image(uploaded_file)
```

```
            pred = model.predict(processed_img, verbose=0)[0][0]

            if pred > 0.5:
                label = "🧬 Tuberculosis Detected"
                confidence = pred
            else:
                label = "✅ Normal"
                confidence = 1 - pred

        st.markdown("---")
        st.subheader("🧠 Prediction Result")
        st.success(label)
        st.markdown(f"**Confidence Level:** `{confidence:.2%}` ")
        st.progress(float(confidence))

else:
    st.info("Please upload a chest X-ray to begin analysis.")

# --- Footer ---
st.markdown('<div class="footer">Model: ResNet50 </div>', unsafe_allow_html=True)
```