

copy-of-untitled14-1-4

November 12, 2025

```
[ ]: import os
import gc
gc.collect()

os.environ['KAGGLE_USERNAME'] = "noraadel"
os.environ['KAGGLE_KEY'] = "76e19273178ad0c217ad1e28c1757c48"

!pip install -q kaggle

!kaggle datasets download -d kmader/skin-cancer-mnist-ham10000

import zipfile

with zipfile.ZipFile("skin-cancer-mnist-ham10000.zip", 'r') as zip_ref:
    zip_ref.extractall("skin_cancer_data")

print(" Dataset downloaded and extracted successfully!")
```

```
[ ]: import pandas as pd, glob
import gc
gc.collect()

meta_path = "skin_cancer_data/HAM10000_metadata.csv"
meta = pd.read_csv(meta_path)
print(meta.head())
meta['image_id'] = meta['image_id'].astype(str) + '.jpg'

img_dirs = glob.glob("skin_cancer_data/*")
print("Possible data folders (showing first 10):")
for d in img_dirs[:10]:
    print(d)
```

```
[ ]: import cv2, numpy as np, os, pandas as pd
from tqdm import tqdm
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import gc
gc.collect()
```

```

meta = pd.read_csv("skin_cancer_data/HAM10000_metadata.csv")

IMG_SIZE = 128
raw_image_folders = [
    "skin_cancer_data/ham10000_images_part_1",
    "skin_cancer_data/ham10000_images_part_2"
]
preproc_dir = "preprocessed_images_aug"
os.makedirs(preproc_dir, exist_ok=True)

# === Hair Removal Function ===
def remove_hair_inpaint(img_rgb):
    gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (17,17))
    blackhat = cv2.morphologyEx(gray, cv2.MORPH_BLACKHAT, kernel)
    _, mask = cv2.threshold(blackhat, 10, 255, cv2.THRESH_BINARY)
    mask = cv2.dilate(mask, cv2.getStructuringElement(cv2.MORPH_RECT, (3,3)),
↳ iterations=1)
    inpainted = cv2.inpaint(cv2.cvtColor(img_rgb, cv2.COLOR_RGB2BGR), mask, 1,
↳ cv2.INPAINT_TELEA)
    inpainted = cv2.cvtColor(inpainted, cv2.COLOR_BGR2RGB)
    return inpainted

# === Data Augmentation Generator ===
datagen = ImageDataGenerator(
    rotation_range=25,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1,
    brightness_range=[0.8, 1.2],
    horizontal_flip=True,
    vertical_flip=True,
    fill_mode='nearest'
)

# === Process and Augment ===
rows = []
count = 0

for idx, row in tqdm(meta.iterrows(), total=len(meta), desc="Preprocessing +
↳ Augmentation"):
    img_name = row['image_id'] + ".jpg"
    found = False
    for d in raw_image_folders:
        p = os.path.join(d, img_name)

```

```

if os.path.exists(p):
    found = True
    img = cv2.imread(p)
    if img is None:
        continue
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = remove_hair_inpaint(img)
    img = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
    img = img.astype('float32') / 255.0

    # Save original preprocessed
    save_path = os.path.join(preproc_dir, img_name)
    cv2.imwrite(save_path, cv2.cvtColor((img*255).astype('uint8'), cv2.
↪COLOR_RGB2BGR))
    rows.append({'image_id': img_name, 'label': row['dx'], 'path': ↪
↪save_path})

    # Generate 3 augmentations per image
    img_expanded = np.expand_dims(img, 0)
    aug_iter = datagen.flow(img_expanded, batch_size=1)

    for i in range(3):
        aug_img = next(aug_iter)[0]
        aug_path = os.path.join(preproc_dir, f"{row['image_id']}_aug{i}.
↪jpg")
        cv2.imwrite(aug_path, cv2.cvtColor((aug_img*255).
↪astype('uint8'), cv2.COLOR_RGB2BGR))
        rows.append({'image_id': f"{row['image_id']}_aug{i}", 'label': ↪
↪row['dx'], 'path': aug_path})
        count += 1
    break

if not found:
    continue

preproc_df = pd.DataFrame(rows)
print(f" Preprocessed + Augmented {count} images. Dataframe shape: {preproc_df.
↪shape}")
preproc_df.head()

```

```

[ ]: import matplotlib.pyplot as plt

def show_before_after(meta_df, raw_folders, preproc_dir, num_samples=5):
    plt.figure(figsize=(10, num_samples * 4))

    for i, row in meta_df.head(num_samples).iterrows():

```

```

img_name = row['image_id'] + ".jpg"
orig_img = None
for d in raw_folders:
    p = os.path.join(d, img_name)
    if os.path.exists(p):
        orig_img = cv2.imread(p)
        if orig_img is not None:
            orig_img = cv2.cvtColor(orig_img, cv2.COLOR_BGR2RGB)
            break

proc_path = os.path.join(preproc_dir, img_name)
proc_img = cv2.imread(proc_path)
if proc_img is not None:
    proc_img = cv2.cvtColor(proc_img, cv2.COLOR_BGR2RGB)

if orig_img is not None and proc_img is not None:
    plt.subplot(num_samples, 2, i*2 + 1)
    plt.imshow(orig_img)
    plt.title(f"Original: {img_name}")
    plt.axis('off')

    plt.subplot(num_samples, 2, i*2 + 2)
    plt.imshow(proc_img)
    plt.title(f"Processed: {img_name}")
    plt.axis('off')

plt.tight_layout()
plt.show()

show_before_after(meta, raw_image_folders, preproc_dir, num_samples=5)

```

```

[ ]: import numpy as np
from tqdm import tqdm
import gc
gc.collect()

def load_all_images(df):
    X = []
    y = []
    for _, r in tqdm(df.iterrows(), total=len(df)):
        p = r['path']
        img = cv2.imread(p) # BGR
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = img.astype('float32') / 255.0
        X.append(img)
        y.append(r['label'])
    return np.array(X), np.array(y)

```

```

X, y = load_all_images(preproc_df)
print("X shape:", X.shape, "y shape:", y.shape)
unique, counts = np.unique(y, return_counts=True)
print("Class distribution:")
for u, c in zip(unique, counts):
    print(u, c)

```

```

[ ]: import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.utils import resample
from tensorflow.keras.utils import to_categorical
import gc

print("Original dataset shape:", X.shape, len(np.unique(y)), "classes")

le = LabelEncoder()
y_enc = le.fit_transform(y)

unique, counts = np.unique(y_enc, return_counts=True)
print("Before balancing:")
for u, c in zip(unique, counts):
    print(f" {le.inverse_transform([u])[0]}: {c}")

from collections import Counter
counter = Counter(y_enc)
max_count = max(counter.values())

X_resampled = []
y_resampled = []

for cls, count in counter.items():
    idx = np.where(y_enc == cls)[0]
    X_cls = X[idx]
    y_cls = y_enc[idx]
    X_up, y_up = resample(X_cls, y_cls, replace=True, n_samples=max_count,
↳ random_state=42)
    X_resampled.append(X_up)
    y_resampled.append(y_up)

X_res = np.vstack(X_resampled)
y_res = np.hstack(y_resampled)

del X_resampled, y_resampled, X_cls, y_cls, X_up, y_up
gc.collect()

y_res_cat = to_categorical(y_res, num_classes=len(le.classes_))

```

```

print("\nAfter balancing:")
unique_res, counts_res = np.unique(y_res, return_counts=True)
for u, c in zip(unique_res, counts_res):
    print(f" {le.inverse_transform([u])[0]}: {c}")

print("\nBalanced dataset shape:", X_res.shape, y_res_cat.shape)

```

```

[ ]: import tensorflow as tf
from tensorflow.keras.applications import VGG16, ResNet50, MobileNetV2,
↳EfficientNetB0
from tensorflow.keras.models import Model
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense, Dropout,
↳Input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score,
↳f1_score
from imblearn.over_sampling import RandomOverSampler
import numpy as np
import pandas as pd
import gc

gc.collect()

#
MODEL_NAME = 'efficientnetb0'
IMG_SHAPE = (IMG_SIZE, IMG_SIZE, 3)
K = 5
EPOCHS = 8
BATCH_SIZE = 32

le = LabelEncoder()
y_enc = le.fit_transform(y)
num_classes = len(le.classes_)
X = tf.image.resize(X, (128, 128)).numpy()

print("      :", num_classes)
print("      :", X.shape)

def get_feature_extractor(model_name, input_shape):
    if model_name == 'vgg16':
        base = VGG16(weights='imagenet', include_top=False,
↳input_shape=input_shape)

```

```

        elif model_name == 'resnet50':
            base = ResNet50(weights='imagenet', include_top=False,
↪input_shape=input_shape)
        elif model_name == 'mobilenetv2':
            base = MobileNetV2(weights='imagenet', include_top=False,
↪input_shape=input_shape)
        elif model_name == 'efficientnetb0':
            base = EfficientNetB0(weights='imagenet', include_top=False,
↪input_shape=input_shape)
        else:
            raise ValueError("Unknown model name")

        x = GlobalAveragePooling2D()(base.output)
        feature_model = Model(inputs=base.input, outputs=x)
        return feature_model

print(" Extracting features...")
feature_model = get_feature_extractor(MODEL_NAME, IMG_SHAPE)
features = feature_model.predict(X, batch_size=32, verbose=1)
print("Features shape:", features.shape)

ros = RandomOverSampler(random_state=42)
X_res, y_res = ros.fit_resample(features, y_enc)
print("Balanced shape:", X_res.shape, y_res.shape)

def create_classifier(input_dim, num_classes):
    inp = Input(shape=(input_dim,))
    x = Dense(256, activation='relu')(inp)
    x = Dropout(0.3)(x)
    out = Dense(num_classes, activation='softmax')(x)
    model = Model(inputs=inp, outputs=out)
    model.compile(optimizer=Adam(learning_rate=1e-4),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# K-Fold Cross Validation
skf = StratifiedKFold(n_splits=K, shuffle=True, random_state=42)
metrics_per_fold = []
fold_no = 1

for train_idx, val_idx in skf.split(X_res, y_res):
    print(f"\n--- Fold {fold_no}/{K} ---")
    X_train, X_val = X_res[train_idx], X_res[val_idx]
    y_train, y_val = y_res[train_idx], y_res[val_idx]

```

```

model = create_classifier(X_train.shape[1], num_classes)

cb = [
    tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3,
↪restore_best_weights=True),
]

history = model.fit(X_train, y_train,
                    validation_data=(X_val, y_val),
                    epochs=EPOCHS,
                    batch_size=BATCH_SIZE,
                    callbacks=cb,
                    verbose=1)

#
y_pred = np.argmax(model.predict(X_val), axis=1)
acc = accuracy_score(y_val, y_pred)
prec = precision_score(y_val, y_pred, average='macro', zero_division=0)
rec = recall_score(y_val, y_pred, average='macro', zero_division=0)
f1 = f1_score(y_val, y_pred, average='macro', zero_division=0)
print(f"Fold {fold_no} - acc: {acc:.4f}, prec: {prec:.4f}, rec: {rec:.4f},
↪f1: {f1:.4f}")

metrics_per_fold.append({'fold': fold_no, 'acc': acc, 'prec': prec, 'rec':
↪rec, 'f1': f1})
model.save(f"classifier_fold_{fold_no}.h5")

fold_no += 1
gc.collect()

#
df_metrics = pd.DataFrame(metrics_per_fold)
print("\nCross-validation results:")
print(df_metrics)
print("\nMean metrics:")
print(df_metrics.mean())

```

```
[ ]: model.save(f"model_fold_{fold_no}.h5")
```

```
[ ]: import os
import numpy as np
from tensorflow.keras.models import load_model
from sklearn.metrics import classification_report

rng = np.random.default_rng(42)
idxs = np.arange(len(X_res))

```

```

test_size = int(0.1 * len(X_res))
test_idx = rng.choice(idxs, size=test_size, replace=False)
X_test_hold = X_res[test_idx]
y_test_hold = y_res[test_idx]

models = []
for i in range(1, K+1):
    path = f"/content/model_fold_6.h5"
    if os.path.exists(path):
        print(f"Loading {path}")
        models.append(load_model(path))
    else:
        print(f"Warning: {path} not found, skipping.")

if len(models) == 0:
    print(" No models found! Skipping ensemble evaluation.")
else:
    print("Loaded", len(models), "models for ensemble.")

X_test_input = X_test_hold

probs = np.zeros((len(X_test_input), len(le.classes_)))
for m in models:
    probs += m.predict(X_test_input, batch_size=32)

probs /= len(models)
y_ens_pred = np.argmax(probs, axis=1)

print(classification_report(y_test_hold, y_ens_pred, target_names=list(le.
↵classes_)))

```

```

[ ]: # Save model of this fold
model.save = f"/content/model_fold_{i}.h5"

fold_no += 1

```

```

[ ]: import os
import numpy as np
from tensorflow.keras.models import load_model
from sklearn.metrics import classification_report, accuracy_score
import matplotlib.pyplot as plt

#
rng = np.random.default_rng(42)
idxs = np.arange(len(X_res))
test_size = int(0.1 * len(X_res))

```

```

test_idx = rng.choice(idxs, size=test_size, replace=False)
X_test_hold = X_res[test_idx]
y_test_hold = y_res[test_idx]

#             ) for)
model1 = load_model("/content/classifier_fold_1.h5")
model2 = load_model("/content/classifier_fold_2.h5")
model3 = load_model("/content/classifier_fold_3.h5")
model4 = load_model("/content/classifier_fold_4.h5")
model5 = load_model("/content/classifier_fold_5.h5")

models = [model1, model2, model3, model4, model5]

#             ) feature_model)
try:
    X_test_input = feature_model.predict(X_test_hold, batch_size=32)
    print(" Using feature_model for feature extraction.")
except:
    X_test_input = X_test_hold
    print(" feature_model not available - using raw test data instead.")

accs = []
probs = np.zeros((len(X_test_input), len(le.classes_)))

print("\n Individual Model Evaluation Reports:")
for i, model in enumerate(models, start=1):
    preds = model.predict(X_test_input, batch_size=32)
    y_pred = np.argmax(preds, axis=1)
    acc = accuracy_score(y_test_hold, y_pred)
    accs.append(acc)
    probs += preds
    print(f"\nModel {i} accuracy: {acc:.4f}")
    print(classification_report(y_test_hold, y_pred, target_names=list(le.
↪classes_)))

probs /= len(models)
y_ens_pred = np.argmax(probs, axis=1)
ensemble_acc = accuracy_score(y_test_hold, y_ens_pred)

print("\n Ensemble Accuracy:", ensemble_acc)
print(classification_report(y_test_hold, y_ens_pred, target_names=list(le.
↪classes_)))

labels = [f"Model {i}" for i in range(1, 6)] + ["Ensemble"]
values = accs + [ensemble_acc]

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

```

```
plt.bar(labels, values)
plt.ylim(0, 1)
plt.title("Model Accuracy Comparison")
plt.ylabel("Accuracy")
plt.xlabel("Model")
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.show()
```