```
"""
Code here is just for reference.
For the final code check, please refer to https://github.com/Ferry-Li/datacv_fr
"""

import os
import cv2
import numpy as np
from insightface.app import FaceAnalysis
from sklearn.metrics.pairwise import cosine_similarity
from tqdm import tqdm
from model import iresnet
import torch
from sklearn.cluster import DBSCAN

def load_images_from_dir(dir_path):
    image_paths = []
    for fname in os.listdir(dir_path):
        if fname.lower().endswith(('.jpg', '.jpeg', '.png', '.bmp', '.webp')):
            image_paths.append(os.path.join(dir_path, fname))
    return sorted(image_paths)

def load_embeddings(image_paths):
    embeddings = []
    valid_image_paths = []

    for path in os.listdir(image_paths):
        if not path.endswith(".npy"):
            continue
        embed_path = os.path.join(image_paths, path)
        idx_name = path.split(".")[0]

        embed = np.load(embed_path)
        embeddings.append(embed)
        valid_image_paths.append(idx_name)

    return embeddings, valid_image_paths

def extract_embeddings(image_paths, fr_model):

    mean = np.array([0.5, 0.5, 0.5])
    std = np.array([0.5, 0.5, 0.5])

    embeddings = []
    valid_image_paths = []

    for path in image_paths:


        img = cv2.imread(path)  # HWC, BGR format
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # Convert to RGB if needed
        # Normalize
        img = ((img / 255.0) - mean) / std  # shape: HWC
        # Convert to CHW
        img = img.transpose(2, 0, 1)  # shape: CHW

        # Convert to float32 tensor
        img = torch.tensor(img, dtype=torch.float32)

        embeddings.append(fr_model(img))
        valid_image_paths.append(path)

    return embeddings, valid_image_paths

def group_by_identity(embeddings, threshold=0.5):
    n = len(embeddings)
    visited = [False] * n
    groups = []

    for i in tqdm(range(n)):
        if visited[i]:
            continue
        group = [i]
        visited[i] = True
        for j in range(i + 1, n):
            sim = cosine_similarity([embeddings[i]], [embeddings[j]])[0][0]
            if sim > threshold and not visited[j]:
                group.append(j)
                visited[j] = True
        groups.append(group)

    return max(groups, key=len) if groups else []

def group_by_identity_dbscan(embeddings, threshold=0.6, min_samples=2):
    """
    Cluster embeddings using DBSCAN and return the largest cluster.
    """
    if isinstance(embeddings[0], torch.Tensor):
        embeddings = [e.detach().cpu().numpy() if e.requires_grad else e.cpu().numpy() for e in embeddings]
    embeddings = np.stack(embeddings)

    # DBSCAN with cosine distance (1 - cosine similarity)
    clustering = DBSCAN(eps=1 - threshold, min_samples=min_samples, metric='cosine')
    labels = clustering.fit_predict(embeddings)

    if np.all(labels == -1):  # all noise
        print("No clusters found.")
        return []

    # Find largest cluster label (excluding noise)
    unique, counts = np.unique(labels[labels != -1], return_counts=True)
    largest_cluster = unique[np.argmax(counts)]

    # Return indices of samples in the largest cluster
    return [i for i, label in enumerate(labels) if label == largest_cluster]

def group_by_identity_dbscan_auto(
    embeddings,
    initial_threshold=0.6,
    min_samples=2,
    min_percent=0.5,
    max_percent=0.8,
    max_iter=10,
    step=0.02
):
    """
    Dynamically adjust DBSCAN threshold to keep the largest cluster within [min_percent, max_percent].
    """
    if isinstance(embeddings[0], torch.Tensor):
        embeddings = [e.detach().cpu().numpy() if e.requires_grad else e.cpu().numpy() for e in embeddings]
    embeddings = np.stack(embeddings)

    n = len(embeddings)
```

```
    threshold = initial_threshold
    best_group = []
    best_percent = 0.0

    for _ in range(max_iter):
        clustering = DBSCAN(eps=1 - threshold, min_samples=min_samples, metric='cosine')
        labels = clustering.fit_predict(embeddings)

        if np.all(labels == -1):
            print(f"No clusters found at threshold {threshold:.2f}")
            threshold += step  # try looser threshold
            continue

        unique, counts = np.unique(labels[labels != -1], return_counts=True)
        largest_cluster = unique[np.argmax(counts)]
        group_indices = [i for i, label in enumerate(labels) if label == largest_cluster]
        percent = len(group_indices) / n

        print(f"Threshold: {threshold:.2f}, Cluster Size: {len(group_indices)}, Percent: {percent:.2%}")

        if min_percent <= percent <= max_percent:
            return group_indices  # good enough

        # Save best so far
        if percent > best_percent:
            best_percent = percent
            best_group = group_indices

        # Adjust threshold
        if percent < min_percent:
            threshold -= step
        elif percent > max_percent:
            threshold += step

        # Clamp between [0.3, 0.9] to avoid extreme values
        threshold = max(0.3, min(0.9, threshold))

    print(f"Best cluster found at ~{best_percent:.2%} of total")
    return best_group



def save_filenames(group_indices, valid_image_paths, dir_path):
    save_txt_root = "fr_training/hsface100k/hsface100k"
    filename = dir_path.split("/")[-1] + '.txt'
    output_name = os.path.join(save_txt_root, filename)

    # Get filenames and sort by numeric value if possible
    selected_files = [os.path.basename(valid_image_paths[i]) for i in group_indices]

    # Sort filenames numerically (e.g., "000001.npy" в†’ 1)
    def extract_index(name):
        return int(os.path.splitext(name)[0])

    selected_files.sort(key=extract_index)

    with open(output_name, "w") as f:
        for filename in selected_files:
            f.write(filename + "\n")

    print(f"Saved to {output_name}: {selected_files}")
    print(f"Total files: {len(selected_files)}")


def main(dir_path):
    print(f"Analyzing images in: {dir_path}")
    image_paths = load_images_from_dir(dir_path)
    have_embeddings = True
    if not image_paths:
        print("No images found.")
        return

    fr_model = iresnet(arch="50", fp16=True)

    if not have_embeddings:
        embeddings, valid_image_paths = extract_embeddings(image_paths, fr_model)
        if not embeddings:
            print("No valid face embeddings extracted.")
            return
    else:
        embeddings, valid_image_paths = load_embeddings(dir_path)

    group = group_by_identity_dbscan_auto(embeddings)
    save_filenames(group, valid_image_paths, dir_path)

if __name__ == "__main__":
    root_dir = "fr_training/hsface100k/hsface100k"
    for ID in tqdm(range(0, 100000)):
        dir_path = os.path.join(root_dir, f"{ID:06d}")
        if os.path.exists(dir_path):
            print(f"Processing {dir_path}...")
            main(dir_path)
        else:
            print(f"Directory {dir_path} does not exist, skipping.")
```

```
"""
Code here is just for reference.
For the final code check, please refer to https://github.com/Ferry-Li/datacv_fr
"""

import os
import json
import re
import time
import base64
from PIL import Image
from openai import OpenAI
from concurrent.futures import ThreadPoolExecutor, as_completed
from tqdm import tqdm

# Explicitly pass API Key
client = OpenAI(api_key="API_KEY", base_url="https://api.chatanywhere.tech/v1")

# Image directory and output result path
image_dir = 'hsface20k'
output_json = 'gpt4o_clean_results.json'

# Check if response is valid (e.g., "001,005,012")
def is_valid_response(response):
    return re.fullmatch(r"(\d{3})(,\d{3})*", response.strip()) is not None

# Send image and prompt to GPT
def ask_identity_check(image_path):
    with open(image_path, "rb") as image_file:
        image_data = image_file.read()
        image_base64 = base64.b64encode(image_data).decode('utf-8')

    messages = [
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": "You will receive an image consisting of several face photos arranged in a grid, where each face has a numeric ID label below it. Please id
                },
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{image_base64}",
                        "detail": "high"
                    }
                }
            ]
        }
    ]

    for _ in range(3):  # Retry up to 3 times
        try:
            response = client.chat.completions.create(
                model="gpt-4o",
                messages=messages,
                max_tokens=50,
                temperature=0
            )
            return response.choices[0].message.content.strip()
        except Exception as e:
            time.sleep(1)
    return None

# Wrapper for processing a single image (used in parallel)
def process_one(image_file):
    image_path = os.path.join(image_dir, image_file)
    key = os.path.splitext(image_file)[0]
    answer = ask_identity_check(image_path)
    return key, answer

# Load existing results (supports resuming)
if os.path.exists(output_json):
    with open(output_json, 'r') as f:
        results = json.load(f)
else:
    results = {}

# Prepare list of unprocessed images
image_files = sorted(f for f in os.listdir(image_dir) if f.endswith('.png'))
pending_files = [f for f in image_files if os.path.splitext(f)[0] not in results]

# Set max number of concurrent threads (e.g., 20)
max_workers = 20

with ThreadPoolExecutor(max_workers=max_workers) as executor:
    future_to_file = {executor.submit(process_one, image_file): image_file for image_file in pending_files}

    for future in tqdm(as_completed(future_to_file), total=len(future_to_file), desc="Processing"):
        image_file = future_to_file[future]
        key = os.path.splitext(image_file)[0]
        try:
            result_key, answer = future.result()
            if answer:
                results[result_key] = answer
                print(f"[в€љ] {image_file} -> {answer}")
            else:
                print(f"[x] {image_file} -> Failed to get response.")
        except Exception as e:
            print(f"[!] Exception processing {image_file}: {e}")

        # Save results in real-time
        with open(output_json, 'w') as f:
            json.dump(results, f, indent=2, ensure_ascii=False)
```

```
"""
Code here is just for reference.
For the final code check, please refer to https://github.com/Ferry-Li/datacv_fr
"""

import os
import random
from PIL import Image
from torchvision import transforms
from tqdm import tqdm

def low_resolution(img, scale_factor=0.5):
    w, h = img.size
    downsampled = img.resize((int(w * scale_factor), int(h * scale_factor)), Image.BILINEAR)
    return downsampled.resize((w, h), Image.BILINEAR)

def get_augmentation_pipeline():
    return transforms.Compose([
        transforms.RandomHorizontalFlip(p=0.5),
        transforms.RandomApply([transforms.ColorJitter(0.4, 0.4, 0.4, 0.1)], p=0.8),
        transforms.RandomGrayscale(p=0.2),
        transforms.RandomApply([
            transforms.RandomAffine(
                degrees=10,
                translate=(0.05, 0.05),
                scale=(0.95, 1.05),
                shear=5
            )
        ], p=0.5),
        transforms.RandomApply([
            transforms.RandomRotation(degrees=5)
        ], p=0.5),
        transforms.GaussianBlur(kernel_size=3, sigma=(0.1, 2.0)),
        transforms.Lambda(lambda img: low_resolution(img, scale_factor=0.5)),
    ])

def cleanup_and_augment(dir_path, target_count=50, log_file="too_few_samples.txt"):
    # txt_path = os.path.join(dir_path, os.path.basename(dir_path) + ".txt")
    id_name = dir_path.split("/")[-1]
    txt_path = os.path.join("fr_training/hsface10k_gpt4o/hsface10k/", f"{id_name}.txt")
    if not os.path.exists(txt_path):
        print(f"ID list file not found: {txt_path}")
        return

    # Step 1: Read allowed image IDs from .txt
    with open(txt_path, "r") as f:
        allowed_ids = set(line.strip().split('.')[0] for line in f if line.strip())

    # Step 2: Log if fewer than 10
    if len(allowed_ids) < 10:
        with open(log_file, "a") as log:
            log.write(os.path.basename(id_name) + "\n")
        print(f"{dir_path}: Only {len(allowed_ids)} images вЂ” logged to {log_file}")

    if len(allowed_ids) < 1:
        print(f"No valid IDs found in {txt_path}, skipping {dir_path}.")
        return

    # Step 3: Remove images not in the allowed list
    image_files = sorted([
        f for f in os.listdir(dir_path)
        if f.lower().endswith('.jpg') and os.path.isfile(os.path.join(dir_path, f))
    ])

    kept_images = []
    for f in image_files:
        id_str = os.path.splitext(f)[0]
        if id_str not in allowed_ids:
            os.remove(os.path.join(dir_path, f))
        else:
            kept_images.append(f)

    print(f"Cleaned up. Kept {len(kept_images)} images.")

    # Step 4: Augment if needed
    if len(kept_images) >= target_count:
        print(f"Already has {len(kept_images)} images (>= {target_count}), skipping augmentation.")
        return

    aug = get_augmentation_pipeline()
    needed = target_count - len(kept_images)
    print(f"Augmenting {needed} more images...")

    base_images = [
        Image.open(os.path.join(dir_path, f)).convert("RGB")
        for f in kept_images
    ]

    used_ids = set(int(os.path.splitext(f)[0]) for f in kept_images)
    next_id = max(used_ids) + 1 if used_ids else 0

    for _ in range(needed):
        base_img = random.choice(base_images)
        aug_img = aug(base_img)
        filename = f"{next_id:03d}.jpg"
        aug_img.save(os.path.join(dir_path, filename))
        next_id += 1

    print(f"Final total: {target_count} images in {dir_path}")

# Example usage
if __name__ == "__main__":
    root_path = "fr_training/hsface10k_gpt4o/hsface10k"

    for ID in tqdm(range(0, 10000)):
        dir_path = os.path.join(root_path, f"{ID:06d}")
        if os.path.exists(dir_path):
            print(f"Processing {dir_path}...")
            cleanup_and_augment(dir_path, target_count=50, log_file="hsface10K_gpt4o_too_few_examples.txt")
        else:
            print(f"Directory {dir_path} does not exist, skipping.")
```

```
"""
Code here is just for reference.
For the final code check, please refer to https://github.com/Ferry-Li/datacv_fr
"""

import os
import cv2
import torch
import numpy as np
from tqdm import tqdm
from PIL import Image
import random

from diffusers.utils import load_image
from diffusers.models import ControlNetModel
from diffusers import LCMScheduler
from diffusers import StableDiffusionXLPipeline

from insightface.app import FaceAnalysis
from pipeline_stable_diffusion_xl_instantid import StableDiffusionXLInstantIDPipeline, draw_kps

# Resize helper
def resize_img(input_image, max_side=1280, min_side=1024, size=None,
               pad_to_max_side=False, mode=Image.BILINEAR, base_pixel_number=64):
    w, h = input_image.size
    if size is not None:
        w_resize_new, h_resize_new = size
    else:
        ratio = min_side / min(h, w)
        w, h = round(ratio*w), round(ratio*h)
        ratio = max_side / max(h, w)
        input_image = input_image.resize([round(ratio*w), round(ratio*h)], mode)
        w_resize_new = (round(ratio * w) // base_pixel_number) * base_pixel_number
        h_resize_new = (round(ratio * h) // base_pixel_number) * base_pixel_number
    input_image = input_image.resize([w_resize_new, h_resize_new], mode)

    if pad_to_max_side:
        res = np.ones([max_side, max_side, 3], dtype=np.uint8) * 255
        offset_x = (max_side - w_resize_new) // 2
        offset_y = (max_side - h_resize_new) // 2
        res[offset_y:offset_y+h_resize_new, offset_x:offset_x+w_resize_new] = np.array(input_image)
        input_image = Image.fromarray(res)
    return input_image

POSES = [
    "facing forward",                  # frontal (less frequent)
    "facing left",                    # strong side view
    "facing right",                   # strong side view
    "head tilted to the left",         # expressive/angled
    "head tilted to the right",        # expressive/angled
    "chin slightly up",               # subtle vertical
    "glancing sideways",              # off-center gaze
]

POSE_WEIGHTS = [
    0.05,  # facing forward (frontal)
    0.25,  # facing left
    0.25,  # facing right
    0.15,  # tilt left
    0.15,  # tilt right
    0.075, # chin up
    0.075  # glancing sideways
]

EXPRESSIONS = [
    "neutral expression", "slight smile", "serious look", "subtle frown",
    "gentle smile", "relaxed face", "thoughtful gaze", "calm demeanor"
]

LIGHTINGS = [
    "under natural daylight", "lit by a soft lamp", "with backlight from a window",
    "in harsh sunlight", "under fluorescent indoor lighting", "with soft shadows",
    "in golden hour lighting", "with overhead lighting"
]

CAMERA_ANGLES = [
    "tight close-up", "medium close-up", "head and shoulders shot",
    "upper-body shot", "zoomed-out portrait", "cropped at chin",
    "high-angle selfie", "low-angle shot"
]

BACKGROUNDS = [
    "in front of a plain wall", "in a cozy room", "in an office setting",
    "with a blurred background", "in a park", "near a window", "with bookshelf behind",
    "at a cafe", "in a hallway", "with bokeh lights in the background"
]

ACCESSORIES = [
    "", "", "",  # empty strings to keep most prompts accessory-free
    " wearing glasses", " with a scarf", " with a beanie hat",
    " wearing a hoodie", " with earphones", " wearing a turtleneck"
]

def generate_aug_prompt():
    pose = random.choices(POSES, weights=POSE_WEIGHTS, k=1)[0]
    expression = random.choice(EXPRESSIONS)
    lighting = random.choice(LIGHTINGS)
    camera = random.choice(CAMERA_ANGLES)
    background = random.choice(BACKGROUNDS)
    accessory = random.choice(ACCESSORIES)

    prompt = (
        f"A {expression} portrait of the person, {pose}, {camera}, "
        f"{lighting}, {background}{accessory}."
    )
    return prompt

def random_transform_face_kps(image, max_scale=1.2, min_scale=0.8, max_shift=0.1):
    """Randomly resize and shift the keypoint image."""
    w, h = image.size
    scale = random.uniform(min_scale, max_scale)
    new_w, new_h = int(w * scale), int(h * scale)

    # Resize
    image = image.resize((new_w, new_h), resample=Image.BILINEAR)

    # Pad to original size with random shift
    pad_w = max(w - new_w, 0)
    pad_h = max(h - new_h, 0)
    offset_x = random.randint(0, pad_w) if pad_w > 0 else 0
    offset_y = random.randint(0, pad_h) if pad_h > 0 else 0

    new_image = Image.new("RGB", (w, h), (255, 255, 255))
```

```
    new_image.paste(image, (offset_x, offset_y))

    return new_image

BASE_IDENTITY_PROMPTS = [
    # Age and Gender
    "a young boy with light skin and freckles, face fully exposed",
    "a teenage girl with tan skin and wavy brown hair, with her face completely visible",
    "an elderly man with a long white beard and glasses, face clearly visible",
    "a middle-aged woman with light skin and straight blonde hair, face fully exposed",
    "a young man with medium brown skin and dreadlocks, with his face uncovered",
    "an elderly woman with dark skin and short curly gray hair, face clearly visible",

    # Ethnicity and Features
    "a South Asian man with a neatly trimmed beard, face exposed and clearly visible",
    "a Hispanic woman with long wavy dark hair, with her face fully exposed",
    "an East Asian person with short straight black hair, face completely visible",
    "a Middle Eastern man with a full beard and a sharp jawline, face fully exposed",
    "a person of Indigenous descent with long braided hair, face clearly visible",
    "a Black woman with medium skin tone and afro-textured hair, face fully exposed",
    "a person of mixed heritage with curly hair and freckles, face clearly visible",

    # Hair Styles and Colors
    "a woman with short platinum blonde hair and bangs, her face fully exposed and visible",
    "a man with long straight black hair tied in a ponytail, face clearly visible",
    "a person with red curly hair and medium skin tone, face fully exposed",
    "a person with blue dyed hair and shaved sides, with their face clearly visible",
    "a man with a bald head and a thick mustache, face fully exposed",
    "a woman with shoulder-length ombre hair and glasses, with her face fully visible",

    # Accessories
    "a person with a round face wearing sunglasses and a hat, with their face clearly visible",
    "a woman with hoop earrings and a nose ring, her face fully exposed",
    "a man with a goatee wearing a baseball cap, face fully exposed",
    "a young person with braces and rectangular glasses, face clearly visible",
    "a person with piercings and colorful hair highlights, face fully exposed",

    # Face Shapes
    "a person with an oval face and light skin, face clearly exposed",
    "a person with a square face and medium brown skin, face fully visible",
    "a person with a heart-shaped face and freckles, face fully exposed",
    "a person with a diamond-shaped face and dark skin, face clearly visible",

    # Expressions
    "a man with a cheerful smile, face fully exposed and visible",
    "a woman with a neutral expression and arched eyebrows, face clearly visible",
    "a young girl laughing with her teeth showing, face fully exposed",
    "a boy with a mischievous grin, face clearly visible",
    "an elderly person with a kind smile and wrinkles, face fully visible",
    "a person with a serious expression and a strong jawline, face exposed",

    # Unique Features
    "a person with vitiligo and short curly hair, face fully exposed",
    "a person with a birthmark on their cheek, face clearly visible",
    "a woman with a beauty mark near her lips, face fully exposed",
    "a man with a scar on his forehead, face clearly visible",
    "a person with heterochromia and medium skin tone, face fully exposed",
    "a person with albinism wearing sunglasses, face clearly visible",

    # Cultural References
    "a South Asian woman wearing a saree and bindi, with her face fully exposed",
    "a Middle Eastern man wearing a keffiyeh and a beard, face fully visible",
    "a person with East Asian features wearing traditional clothing, face clearly visible",
    "a Black man with a shaved head and a dashiki, face fully exposed",
    "a Hispanic woman with a flower crown in her hair, face clearly visible",
    "an Indigenous person with traditional face paint and braids, face fully visible",

    # General
    "a person with long curly hair and a warm smile, face fully exposed",
    "a young man with a round face and freckles, face clearly visible",
    "a woman with light skin and short spiky hair, face fully exposed",
    "a man with medium skin tone and a chiseled jawline, face clearly visible",
    "a teenager with braces and curly dark brown hair, face fully exposed",
    "a person with a medium-length beard and light brown eyes, face clearly visible",
    "a child with dark skin and a big grin, face fully exposed",
]

if __name__ == "__main__":

    import argparse
    parser = argparse.ArgumentParser(description="Generate 10K unique faces with InstantID")
    parser.add_argument("--id_start", type=int, default=0, help="Start ID number (inclusive)")
    parser.add_argument("--id_end", type=int, default=10000, help="End ID number (exclusive)")
    args = parser.parse_args()
    # Load face detection model
    app = FaceAnalysis(name='antelopev2', root='./', providers=['CUDAExecutionProvider', 'CPUExecutionProvider'])
    app.prepare(ctx_id=0, det_size=(640, 640))

    # Load ControlNet and InstantID pipeline
    controlnet = ControlNetModel.from_pretrained("./checkpoints/ControlNetModel", torch_dtype=torch.float16)
    pipe = StableDiffusionXLInstantIDPipeline.from_pretrained(
        "sd-xl-base-1.0",
        controlnet=controlnet,
        torch_dtype=torch.float16
    )
    pipe.cuda()

    # Load adapters
    pipe.load_lora_weights("./checkpoints/pytorch_lora_weights.safetensors")
    pipe.fuse_lora()
    pipe.load_ip_adapter_instantid("./checkpoints/ip-adapter.bin")
    pipe.scheduler = LCMScheduler.from_config(pipe.scheduler.config)

    # Load base pipeline
    base_pipe = StableDiffusionXLPipeline.from_pretrained(
        "sd-xl-base-1.0",
        torch_dtype=torch.float16
    ).to("cuda")
    base_pipe.enable_model_cpu_offload()

    # Output folders
    OUTPUT_DIR = "./sd_xl_ids"
    CROP_DIR = "./sd_xl_crop_ids"
    os.makedirs(OUTPUT_DIR, exist_ok=True)
    os.makedirs(CROP_DIR, exist_ok=True)

    n_prompt = "(lowres, low quality, worst quality:1.2), (text:1.2), watermark, painting, drawing, illustration, glitch, deformed, mutated, cross-eyed, ugly, disfigur

    target_faces = args.id_end
    identity_idx = args.id_start
    pbar = tqdm(total=target_faces, desc="Generating 10K Unique Faces", initial=identity_idx)

    while identity_idx < target_faces:
```

```
seed = random.randint(0, 999999)
generator = torch.manual_seed(seed)

base_prompt = random.choice(BASE_IDENTITY_PROMPTS)
base_image = base_pipe(
    prompt=base_prompt,
    negative_prompt=n_prompt,
    guidance_scale=7.5,
    num_inference_steps=30,
    generator=generator
).images[0]

resized_face = resize_img(base_image)
face_info = app.get(cv2.cvtColor(np.array(resized_face), cv2.COLOR_RGB2BGR))

if len(face_info) == 0:
    print(f"No face detected. Skipping.")
    continue

face_info = sorted(face_info, key=lambda x: (x['bbox'][2]-x['bbox'][0])*(x['bbox'][3]-x['bbox'][1]))[-1]
face_emb = face_info['embedding']

face_kps = draw_kps(resized_face, face_info['kps'])
face_kps = random_transform_face_kps(face_kps)

aug_prompt = generate_aug_prompt()
controlnet_conditioning_scale = random.uniform(0.1, 0.5)
ip_adapter_scale = random.uniform(0.95, 1.0)

aug_image = pipe(
    prompt=aug_prompt,
    negative_prompt=n_prompt,
    image_embeds=face_emb,
    image=face_kps,
    controlnet_conditioning_scale=controlnet_conditioning_scale,
    ip_adapter_scale=ip_adapter_scale,
    num_inference_steps=10,
    guidance_scale=1.0,
    generator=generator
).images[0]

id_dir_full = os.path.join(OUTPUT_DIR, f"ID_{identity_idx:05d}")
id_dir_crop = os.path.join(CROP_DIR, f"ID_{identity_idx:05d}")
os.makedirs(id_dir_full, exist_ok=True)
os.makedirs(id_dir_crop, exist_ok=True)

full_path = os.path.join(id_dir_full, "0.jpg")
aug_image.save(full_path)

try:
    detected_faces = app.get(cv2.cvtColor(np.array(aug_image), cv2.COLOR_RGB2BGR))
    if detected_faces:
        best_face = sorted(detected_faces, key=lambda x: (x['bbox'][2]-x['bbox'][0])*(x['bbox'][3]-x['bbox'][1]))[-1]
        x1, y1, x2, y2 = map(int, best_face['bbox'])
        cropped_face = aug_image.crop((x1, y1, x2, y2))
        crop_path = os.path.join(id_dir_crop, "0.jpg")
        cropped_face.save(crop_path)

        # Increment only if face is successfully cropped and saved
        identity_idx += 1
        pbar.update(1)

    else:
        print(f"No face detected in final image. Skipping.")
except Exception as e:
    print(f"Error cropping face: {e}")
```

```
"""
Code here is just for reference.
For the final code check, please refer to https://github.com/Ferry-Li/datacv_fr
"""

import os
import random
from collections import defaultdict


def load_txt(file_path):
    with open(file_path, 'r') as f:
        return [line.strip() for line in f if line.strip()]


def extract_id_original(image_path):
    # Extract ID like 000466 from: /.../hsface10k/000466/008.jpg
    return os.path.basename(os.path.dirname(image_path))


def extract_id_synthetic(image_path):
    # Extract ID like ID_05669 from: /.../generated_images_ref/ID_05669/003.jpg
    return os.path.basename(os.path.dirname(image_path))


def create_mixed_dataset(
    original_paths_txt,
    too_few_ids_txt,
    synthetic_paths_txt,
    output_txt
):
    # Load data
    original_paths = load_txt(original_paths_txt)
    too_few_ids = set(load_txt(too_few_ids_txt))
    synthetic_paths = load_txt(synthetic_paths_txt)

    print(f"Original images loaded: {len(original_paths)}")
    print(f"Too-few IDs to remove: {len(too_few_ids)}")
    print(f"Synthetic images loaded: {len(synthetic_paths)}")

    # Step 1: Filter out original paths with too-few IDs
    filtered_original = [
        path for path in original_paths
        if extract_id_original(path) not in too_few_ids
    ]
    num_removed_images = len(original_paths) - len(filtered_original)
    print(f"Remaining original images: {len(filtered_original)} ({num_removed_images} removed)")

    # Step 2: Group synthetic paths by ID
    id_to_synth_paths = defaultdict(list)
    for path in synthetic_paths:
        synth_id = extract_id_synthetic(path)
        id_to_synth_paths[synth_id].append(path)

    # Step 3: For each removed ID, randomly pick one synthetic ID
    removed_ids_count = len(too_few_ids)
    available_synthetic_ids = list(id_to_synth_paths.keys())
    random.shuffle(available_synthetic_ids)

    selected_synthetic_paths = []
    used_ids = set()

    for _ in range(removed_ids_count):
        if not available_synthetic_ids:
            print("Not enough synthetic IDs to fully make up for removed IDs.")
            break
        synth_id = available_synthetic_ids.pop()
        used_ids.add(synth_id)
        selected_synthetic_paths.extend(id_to_synth_paths[synth_id])

    print(f"Used {len(used_ids)} synthetic IDs")
    print(f"Total synthetic images added: {len(selected_synthetic_paths)}")

    # Step 4: Mix synthetic and original (preserve grouping)
    final_list = selected_synthetic_paths + filtered_original

    # Step 5: Save to output
    with open(output_txt, "w") as f:
        for line in final_list:
            f.write(line + "\n")

    print(f"Final dataset written to: {output_txt}")
    print(f"Total images in final list: {len(final_list)}")

if __name__ == "__main__":
    create_mixed_dataset(
        original_paths_txt="hsface10K_makeup.txt",
        too_few_ids_txt="hsface10K_too_few_examples.txt",
        synthetic_paths_txt="sd_gen_id.txt",
        output_txt="hsface10K_gpt4o_sd.txt"
    )
```