User:

As a data science student working with the CRISP-DM methodology, I have a dataset of 4,000+ flower images (classes: daisy, tulip, rose, sunflower, etc.). Please provide Python code to build a deep learning model for flower classification following the CRISP-DM methodology. Since I have limited CPU and resources, please make sure the solution is efficient. Break the solution into small, manageable chunks that I can follow step by step. After each chunk, wait for me to say 'continue' before moving to the next part. The code should be modular, well-commented, and ready to run in a Jupyter notebook. Make sure each chunk clearly indicates which CRISP-DM phase it belongs to. Kaggle flower data set: import kagglehub path = kagglehub.dataset_download("alxmamaev/flowers-recognition") print("Path to dataset files:", path) Ensure that every step follows best practices in deep learning (data augmentation, transfer learning, regularization, and evaluation) to achieve the highest possible accuracy while staying resource-efficient.

ChatGPT:

Awesome—let’s tackle this CRISP-DM style in small, digestible chunks.
I’ll start with **Chunk 1** (Business Understanding + Project Setup). When you’re ready for the next part, just say **“continue”**.

CHUNK 1 — Business Understanding & Project Setup (CRISP-DM: Business Understanding)

What we’re doing in this chunk

- Frame the problem and success criteria.
- Set up a lightweight, reproducible environment for resource-constrained training.
- Download the Kaggle dataset using your snippet and prepare a simple project structure.

Design choices for efficiency

- **Transfer learning** (pretrained EfficientNet-B0 or MobileNetV3-Small) to minimize training compute.
- **Smaller input size (224×224)** and **small batch** to keep memory low on CPU.
- **Mixed precision** if a GPU/MPS is available (safely falls back to full precision on CPU).
- Clean **config** object so you can tweak hyperparams in one place later.

---

1) Problem framing & success metric (markdown cell)

> **Goal:** Build a flower image classifier (daisy, dandelion, rose, sunflower, tulip) with strong accuracy using limited compute.
> **Primary metric:** Top-1 accuracy on a held-out test set; we’ll also track macro-F1 to account for any class imbalance.
> **Constraints:** Prefer CPU-friendly training; gracefully leverage GPU/MPS if present.
> **Strategy:** Transfer learning + data augmentation + regularization (label smoothing, weight decay, early stopping).

(Put the above as a markdown cell in your notebook if you like.)

---

2) Environment & dataset bootstrap (code cell)

> Paste the whole block below into one Jupyter cell.

```
python
# ==========================
# CRISP-DM: Business Understanding / Setup
# Project bootstrap: imports, device check, reproducibility, dataset download
# ==========================

# Standard libs
import os, sys, random, json, shutil
from pathlib import Path
import warnings
warnings.filterwarnings("ignore")
```

```
# Repro & numerics
import numpy as np

# Deep learning
import torch

# Vision utils (we'll use these later; ok to import now)
import torchvision
from torchvision import transforms

# Visualization (used later)
import matplotlib.pyplot as plt

# Dataset download via KaggleHub (as you provided)
import kagglehub

# --------------------------
# 1) Reproducibility helpers
# --------------------------
def set_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    # Make some ops deterministic when possible (may slightly reduce speed)
    torch.backends.cudnn.deterministic = False
    torch.backends.cudnn.benchmark = True  # Good for performance when input sizes are constant

set_seed(42)

# --------------------------
# 2) Device discovery
# --------------------------
def get_device():
    if torch.cuda.is_available():
        return torch.device("cuda")
    # Apple Silicon (Metal) support
    if hasattr(torch.backends, "mps") and torch.backends.mps.is_available():
        return torch.device("mps")
    return torch.device("cpu")

DEVICE = get_device()

def env_summary():
    info = {
        "python": sys.version.split()[0],
        "torch_version": torch.__version__,
        "torchvision_version": torchvision.__version__,
        "device": str(DEVICE),
        "cuda_available": torch.cuda.is_available(),
        "num_cpu_threads": os.cpu_count(),
    }
    print(json.dumps(info, indent=2))

print("== Environment Summary ==")
env_summary()

# --------------------------
# 3) Project directories
# --------------------------
ROOT = Path(".").resolve()
ARTIFACTS_DIR = ROOT / "artifacts"
ARTIFACTS_DIR.mkdir(parents=True, exist_ok=True)  # for checkpoints, logs, etc.
print(f"Artifacts will be saved to: {ARTIFACTS_DIR}")

# --------------------------
# 4) Download dataset (KaggleHub)
#    Dataset: alxmamaev/flowers-recognition
#    Structure: <download_path>/flowers/<class_name>/*.jpg
# --------------------------
print("Downloading dataset via kagglehub (first run may take a bit depending on your connection)...")
ds_path = kagglehub.dataset_download("alxmamaev/flowers-recognition")
print("Path to dataset files:", ds_path)

# The actual images are inside a 'flowers' subfolder in this dataset
DATASET_DIR = Path(ds_path) / "flowers"
if not DATASET_DIR.exists():
    # Fallback: if structure changes, just use the root as-is
    DATASET_DIR = Path(ds_path)

print(f"Resolved dataset directory: {DATASET_DIR}")

# Quick sanity check of class folders
if DATASET_DIR.exists():
    class_dirs = [p.name for p in DATASET_DIR.iterdir() if p.is_dir()]
    print(f"Found class folders: {sorted(class_dirs)}")
    # Count a few samples per class (lightweight)
    counts = {c: len(list((DATASET_DIR / c).glob("*"))) for c in class_dirs}
    print("Sample counts (first few):", dict(list(counts.items())[:10]))
else:
    raise FileNotFoundError("Dataset directory not found. Please verify the KaggleHub path.")

# --------------------------
# 5) Global configuration (tweak later as needed)
# --------------------------
from dataclasses import dataclass

@dataclass
class Config:
    dataset_dir: Path = DATASET_DIR
    artifacts_dir: Path = ARTIFACTS_DIR
    image_size: int = 224                # smaller => faster & lighter
    batch_size: int = 16                 # start small for CPU, we can tune later
    num_workers: int = max(1, (os.cpu_count() or 2) // 2)  # conservative for notebooks
    model_name: str = "efficientnet_b0"  # or "mobilenet_v3_small" (even lighter)
    learning_rate: float = 3e-4
    weight_decay: float = 1e-4
    label_smoothing: float = 0.1
    epochs: int = 10                     # placeholder; early stopping will guard overfit
    freeze_backbone_epochs: int = 2      # warmup with frozen backbone
    mixed_precision: bool = True         # autocast if GPU/MPS is available

CFG = Config()
print("Config:", CFG)

# Save config snapshot for reproducibility
with open(CFG.artifacts_dir / "config.json", "w") as f:
    json.dump({k: str(v) if isinstance(v, Path) else v for k, v in CFG.__dict__.items()}, f, indent=2)
print(f"Saved config to {CFG.artifacts_dir / 'config.json'}")
```

What's next

In **Chunk 2 (Data Understanding)** we will:

- Create a clean **train/val/test split** (stratified).
- Inspect **class distribution** and **image shapes**.
- Visualize a small **grid of samples**.
- Compute a lightweight **dataset summary** so we can choose augments & model.

✅ Chunk 1 — quick recap of your requirements

- Provide efficient, modular, well-commented code ready for Jupyter.
- Follow CRISP-DM and explicitly label the phase.
- Use KaggleHub to fetch the flower dataset.
- Keep resource use low and set up for transfer learning, augmentation, and regularization later.

🧭 Mindmap update (CRISP-DM)

- **Business Understanding:** ✅ Defined goal, constraints, metrics, and high-level plan.
- **Data Understanding:** ⏳ Up next (splits, EDA, visuals).
- **Data Preparation:** later (augmentations & loaders).
- **Modeling:** later (transfer learning w/ EfficientNet/MobileNet).
- **Evaluation:** later (accuracy, macro-F1, confusion matrix).
- **Deployment:** later (export, simple inference script).

Say **“continue”** when you’re ready for **Chunk 2 (Data Understanding)**.

User:

let's continue

ChatGPT:

Great—here’s **Chunk 2** (Data Understanding). When you’re ready to proceed, say **“continue”**.

CHUNK 2 — Data Understanding (CRISP-DM: Data Understanding)

What we’ll do in this chunk

- Build a **manifest** of images and labels (no file copying).
- Make a **stratified train/val/test split** and save CSVs.
- Quick **EDA**: class distribution & basic image dimension stats (on a safe sample).
- Visualize a **small grid** of training samples.

  Paste the entire cell below into your notebook.

```python
python
# ==========================
# CRISP-DM: Data Understanding
# Build manifest, stratified splits, light EDA, and sample visualization
# ==========================
from pathlib import Path
import random, json
from collections import Counter, defaultdict

import pandas as pd
import numpy as np
from PIL import Image, UnidentifiedImageError
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# ---------- Helpers ----------
def list_images_by_class(root: Path, exts={".jpg", ".jpeg", ".png", ".bmp"}):
    """Scan <root>/<class_name>/* and return list of (path, class_name)."""
    records = []
    classes = []
    for d in sorted(p for p in root.iterdir() if p.is_dir()):
        cls = d.name
        classes.append(cls)
        for p in d.iterdir():
            if p.is_file() and p.suffix.lower() in exts:
                records.append((str(p), cls))
    return records, sorted(classes)

def make_manifest(dataset_dir: Path) -> pd.DataFrame:
    recs, classes = list_images_by_class(dataset_dir)
    if not recs:
        raise RuntimeError(f"No images found under {dataset_dir}.")
    label_to_idx = {c: i for i, c in enumerate(classes)}
    df = pd.DataFrame(recs, columns=["path", "label"])
    df["label_idx"] = df["label"].map(label_to_idx)
    return df, classes, label_to_idx

def stratified_splits(df: pd.DataFrame, train=0.70, val=0.15, test=0.15, seed=42):
    assert abs(train + val + test - 1.0) < 1e-6, "Splits must sum to 1.0"
    # First split train vs temp
    df_train, df_temp = train_test_split(
        df, test_size=(1.0 - train), stratify=df["label_idx"], random_state=seed
    )
    # Split temp into val and test
    rel_val = val / (val + test)
    df_val, df_test = train_test_split(
        df_temp, test_size=(1.0 - rel_val), stratify=df_temp["label_idx"], random_state=seed
    )
    return df_train.reset_index(drop=True), df_val.reset_index(drop=True), df_test.reset_index(drop=True)

def summarize_counts(df: pd.DataFrame, title: str):
    counts = df["label"].value_counts().sort_index()
    print(f"\n== {title} ==")
    print(counts.to_string())
    print("\nClass proportions:")
    print((counts / counts.sum()).round(3).to_string())

def robust_image_open(path: str):
    """Open image safely and return PIL.Image or None if corrupted."""
    try:
        img = Image.open(path).convert("RGB")
        return img
    except (UnidentifiedImageError, OSError):
        return None

def sample_image_stats(df: pd.DataFrame, max_per_class=100, seed=42):
    """Compute lightweight width/height stats on a small sample (CPU friendly)."""
    rng = np.random.default_rng(seed)
    sampled = []
    for cls, sub in df.groupby("label"):
        use = min(max_per_class, len(sub))
        sampled.append(sub.sample(n=use, random_state=seed))
    sampled_df = pd.concat(sampled, ignore_index=True)
    widths, heights, bad = [], [], 0
    for p in sampled_df["path"]:
        img = robust_image_open(p)
        if img is None:
            bad += 1
            continue
        w, h = img.size  # PIL gives (W, H)
        widths.append(w); heights.append(h)
    def stats(arr):
        arr = np.array(arr)
        if arr.size == 0:
            return {}
        return {
            "count": int(arr.size),
            "min": int(arr.min()),
            "p05": float(np.percentile(arr, 5)),
            "median": float(np.median(arr)),
            "mean": float(arr.mean()),
            "p95": float(np.percentile(arr, 95)),
            "max": int(arr.max()),
        }
    return {
        "width": stats(widths),
        "height": stats(heights),
        "corrupted_or_unreadable": int(bad),
        "sampled_images": int(sum(len(s) for _, s in df.groupby("label")) if df is not None else 0)
    }

def plot_class_distribution(df: pd.DataFrame, title="Class distribution"):
    counts = df["label"].value_counts().sort_index()
    plt.figure(figsize=(6, 3.5))
    counts.plot(kind="bar")
    plt.title(title)
    plt.xlabel("Class")
    plt.ylabel("Count")
    plt.tight_layout()
    plt.show()

def show_sample_grid(df: pd.DataFrame, n_per_class=3, seed=42):
    """Show a small grid of images per class from df."""
    rng = random.Random(seed)
    groups = df.groupby("label")
    samples = []
    for cls, sub in groups:
        pick = sub.sample(n=min(n_per_class, len(sub)), random_state=seed)
        for _, row in pick.iterrows():
            samples.append((row["path"], cls))
    if not samples:
        print("No samples to display.")
        return
    cols = n_per_class
    classes = sorted(df["label"].unique())
    rows = len(classes)
    plt.figure(figsize=(cols * 2.4, rows * 2.4))
    i = 1
    for cls in classes:
        sub = [s for s in samples if s[1] == cls]
        for j, (p, lbl) in enumerate(sub):
            img = robust_image_open(p)
            if img is None:  # skip corrupted
                continue
            plt.subplot(rows, cols, i)
            plt.imshow(img)
            plt.axis("off")
            plt.title(lbl, fontsize=9)
            i += 1
    plt.tight_layout()
    plt.show()

# ---------- Execute Data Understanding pipeline ----------
print("Scanning dataset and building manifest...")
df_all, classes, label_to_idx = make_manifest(CFG.dataset_dir)
print(f"Classes ({len(classes)}): {classes}")
print(f"Total images found: {len(df_all)}")

# Save manifest for reproducibility
manifest_path = CFG.artifacts_dir / "manifest.csv"
df_all.to_csv(manifest_path, index=False)
print(f"Saved manifest to {manifest_path}")
```

```
# Stratified splits
print("\nCreating stratified train/val/test splits (70/15/15)...")
train_df, val_df, test_df = stratified_splits(df_all, train=0.70, val=0.15, test=0.15, seed=42)

# Save split CSVs
train_csv = CFG.artifacts_dir / "train.csv"
val_csv   = CFG.artifacts_dir / "val.csv"
test_csv  = CFG.artifacts_dir / "test.csv"
train_df.to_csv(train_csv, index=False)
val_df.to_csv(val_csv, index=False)
test_df.to_csv(test_csv, index=False)
print(f"Saved splits:\n  {train_csv}\n  {val_csv}\n  {test_csv}")

# Summaries
summarize_counts(df_all, "Overall class counts")
summarize_counts(train_df, "Train class counts")
summarize_counts(val_df, "Val class counts")
summarize_counts(test_df, "Test class counts")

# Plot distribution (overall)
plot_class_distribution(df_all, title="Overall class distribution")

# Lightweight image dimension stats (safe sample, ~<=100 per class)
print("\nComputing lightweight image dimension stats (sampled)...")
img_stats = sample_image_stats(df_all, max_per_class=100, seed=42)
stats_path = CFG.artifacts_dir / "image_stats.json"
with open(stats_path, "w") as f:
    json.dump(img_stats, f, indent=2)
print("Image stats (sampled):")
print(json.dumps(img_stats, indent=2))
print(f"Saved image stats to {stats_path}")

# Visual sanity-check: a few training images per class
print("\nShowing a small grid of training samples (may take a moment on CPU)...")
show_sample_grid(train_df, n_per_class=3, seed=42)
```

Notes on efficiency

- We **don't copy files**—we use CSV manifests that point to original paths.
- Stats are computed on a **small per-class sample** to avoid heavy I/O.
- Plots are minimal and quick; safe to skip if your environment is very tight.

What's next

In **Chunk 3 (Data Preparation)** we will:

- Define **PyTorch datasets & dataloaders** from the CSVs (no copying).
- Add **data augmentation** (RandomResizedCrop, flips, color jitter).
- Decide on **normalization** (ImageNet stats vs. computed stats).
- Wire up a clean **config-driven** input pipeline.

✅ Chunk 2 — quick recap of your requirements

- Perform stratified **train/val/test split** and save artifacts.
- Provide **light EDA** (counts, proportions, dimensions).
- **Visualize** a few samples to verify labels and data quality.
- Keep everything **CPU/resource-friendly** and modular.

🧭 Mindmap update (CRISP-DM)

- **Business Understanding:** ✅ done.
- **Data Understanding:** ✅ manifest, splits, EDA, visuals complete.
- **Data Preparation:** ⏳ next (augmentations, datasets, loaders).
- **Modeling:** later (transfer learning + regularization).
- **Evaluation:** later (accuracy, macro-F1, confusion matrix).
- **Deployment:** later (export & inference script).

Say **“continue”** when you’re ready for **Chunk 3 (Data Preparation)**.

User:

Before proceeding please explain the EDA process that is followed

---

ChatGPT:

Totally—here’s what the **EDA (Exploratory Data Analysis)** in **Chunk 2 — Data Understanding** actually did and why it matters for CRISP-DM.

EDA Overview (CRISP-DM: Data Understanding)

1) Dataset integrity & structure

- **Scan folder layout** `<root>/<class_name>/*.jpg` to ensure expected classes exist.
- **Collect file paths + labels** and verify we actually have images per class.
- **Why:** Confirms basic data health before we invest time in modeling; surfaces missing folders or empty classes early.

2) Manifest creation (reproducible indexing)

- Build a **DataFrame manifest** with `path`, `label`, and `label_idx`, and save it to `artifacts/manifest.csv`.
- **Why:** A manifest makes experiments reproducible, avoids copying files, and is a single source of truth for splits and loaders later.

3) Stratified train/val/test split (70/15/15)

- Use **stratification on label_idx** to maintain class proportions across splits; save `train.csv`, `val.csv`, `test.csv`.
- **Leakage control:** Splits are done at the **file level** once—no overlap between sets; augmentation is applied **only** to train later.
- **Why:** Keeps validation/test representative and prevents optimistic metrics due to leakage.

4) Class balance summary

- Print **counts and proportions** per class for overall/train/val/test; plot a quick bar chart.
- **Why:** Detects imbalance; if severe, we’d consider **class-weighted loss** or **balanced sampling**. Even mild imbalance can skew accuracy without macro-F1.

5) Lightweight image dimension & quality stats (sampled)

- Randomly sample up to **100 images per class**, open robustly (skip unreadable), and compute **width/height** stats (min/median/mean/p95/max).
- **Corruption check:** Count unreadable images (if any).
- **Why:** Guides **input size** (we use 224×224 for efficiency), **crop/resize strategy** (RandomResizedCrop keeps aspect ratio), and flags any systematic class resolution differences.

6) Visual sanity check (small grid per class)

- Display a **tiny grid** (e.g., 3 images/class) from the **training** set.

- **Why:** Quick human-in-the-loop verification for **mislabels, duplicates, odd crops/orientations**, background biases (e.g., sunflowers always outdoors).

7) Artifacting for traceability

- Persist **manifest**, **splits**, and **image_stats.json** under `/artifacts`.
- **Why:** Ensures we can reproduce results, compare runs, and debug issues later (CRISP-DM emphasizes traceable iterations).

How these EDA findings influence the next steps

- **Augmentations:**
  - If images vary in scale/aspect: use **RandomResizedCrop(224)**.
  - Natural photos → **HorizontalFlip**, light **ColorJitter**, and small **RandomRotation** help generalization.
- **Normalization:**
  - Use **ImageNet mean/std** (fits transfer learning). We can compute dataset stats later if desired, but ImageNet is usually fine and faster.
- **Class imbalance (if observed):**
  - Enable **class-weighted CrossEntropy** or **WeightedRandomSampler**.
- **Resource constraints:**
  - Keep **batch_size** small (e.g., 16) and image size 224; prefer **EfficientNet-B0** or **MobileNetV3-Small**; enable mixed precision only when GPU/MPS is available.

Optional (if you want to go deeper later)

- **Duplicate detection:** perceptual hashing to prune near-duplicates (saves overfitting).
- **Color/brightness histograms:** spot domain biases; may tune ColorJitter ranges.
- **Per-class resolution profiles:** if one class has tiny images, consider **center-crop** + **mild sharpening** (still light-weight).
- **Train/val leakage audit:** hash filenames or IDs to ensure no overlap.
- **Compute dataset mean/std:** nice-to-have; not necessary for TF transfer learning with ImageNet backbones.

✅ Quick recap of your request (for this chunk)

- Explain the **EDA process** we followed: structure check, manifest, **stratified splits**, class balance summaries, **sampled image stats**, and **visual sanity checks**, plus how this informs augmentations, normalization, and efficiency choices.

🧭 Mindmap update (CRISP-DM)

- **Business Understanding:** ✅ done.
- **Data Understanding:** ✅ EDA explained; artifacts saved.
- **Data Preparation:** ⏳ next—implement datasets, augmentations, and dataloaders based on these findings.
- **Modeling:** later—transfer learning with EfficientNet/MobileNet + regularization.
- **Evaluation:** later—accuracy, macro-F1, confusion matrix.
- **Deployment:** later—export + simple inference notebook cell.

Say **"continue"** when you want me to deliver **Chunk 3 (Data Preparation)**.

User:

Great let's continue

ChatGPT:

Awesome—here's **Chunk 3** (Data Preparation). When you're ready for the next part, say **"continue."**

CHUNK 3 — Data Preparation (CRISP-DM: Data Preparation)

What we'll do now

- Define **transforms & augmentations** optimized for small/CPU setups.
- Implement a **CSV-driven PyTorch dataset** (no file copying).
- Build **DataLoaders** with reproducibility, optional **class-weighted sampling**, and sensible performance flags.
- (Optional) **Preview** a small augmented batch.

  Paste the whole block below into **one** Jupyter cell.

```
python
# ==========================
# CRISP-DM: Data Preparation
# Transforms, Dataset, Sampler, and DataLoaders
# ==========================
from dataclasses import asdict
from pathlib import Path
import json, random
from typing import Tuple, Dict, Optional

import pandas as pd
import numpy as np
from PIL import Image

import torch
from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler
from torchvision import transforms

# Reuse CFG, DEVICE from earlier chunks
assert 'CFG' in globals(), "Config (CFG) not found. Please run Chunk 1 first."
assert (CFG.artifacts_dir / "train.csv").exists(), "Missing train/val/test CSVs. Please run Chunk 2."

# --------------------------
# 1) Transforms & Augmentations
# --------------------------
IMAGENET_MEAN = (0.485, 0.456, 0.406)
IMAGENET_STD  = (0.229, 0.224, 0.225)

def build_transforms(image_size: int = 224, normalize: str = "imagenet"):
    """
    Returns torchvision transforms for train/eval.
    Designed to be light but effective for natural images on small compute.
    """
    if normalize == "imagenet":
        norm_mean, norm_std = IMAGENET_MEAN, IMAGENET_STD
    else:
        # Placeholder for custom stats; ImageNet is usually best for transfer learning speed.
        norm_mean, norm_std = IMAGENET_MEAN, IMAGENET_STD

    train_tfms = transforms.Compose([
        transforms.RandomResizedCrop(image_size, scale=(0.7, 1.0), ratio=(0.75, 1.33)),
        transforms.RandomHorizontalFlip(p=0.5),
        transforms.ColorJitter(brightness=0.15, contrast=0.15, saturation=0.10, hue=0.02),
        transforms.RandomRotation(degrees=10),
        transforms.ToTensor(),
        transforms.Normalize(norm_mean, norm_std),
    ])

    eval_tfms = transforms.Compose([
        transforms.Resize(int(image_size * 1.15)),
        transforms.CenterCrop(image_size),
        transforms.ToTensor(),
        transforms.Normalize(norm_mean, norm_std),
    ])
    return train_tfms, eval_tfms

# --------------------------
# 2) CSV-driven Dataset
# --------------------------
class FlowerCsvDataset(Dataset):
    """
    Lightweight dataset that reads image paths/labels from a CSV.
    Expects columns: ['path', 'label', 'label_idx'] as created in Chunk 2.
    """
    def __init__(self, csv_path: Path, transform=None):
        self.df = pd.read_csv(csv_path)
        # minimal validation
        for col in ("path", "label", "label_idx"):
            if col not in self.df.columns:
                raise ValueError(f"CSV at {csv_path} missing column: {col}")
        self.paths = self.df["path"].tolist()
        self.targets = self.df["label_idx"].astype(int).tolist()
        self.labels = self.df["label"].tolist()
        self.transform = transform

    def __len__(self):
        return len(self.paths)

    def __getitem__(self, idx: int):
        path = self.paths[idx]
        target = self.targets[idx]
        # Robust load (RGB)
        with Image.open(path) as img:
            img = img.convert("RGB")
        if self.transform is not None:
            img = self.transform(img)
        return img, target

# --------------------------
# 3) Reproducible workers & class weights
# --------------------------
def seed_worker(worker_id):
    # Ensures dataloader workers are deterministically seeded
    worker_seed = torch.initial_seed() % 2**32
    np.random.seed(worker_seed)
    random.seed(worker_seed)

def compute_class_weights(train_csv: Path) -> Dict[int, float]:
    """
    Inverse-frequency weights per class (for optional WeightedRandomSampler).
    """
    df = pd.read_csv(train_csv)
    counts = df["label_idx"].value_counts().sort_index()
    total = counts.sum()
    # Classic inverse frequency; scale so mean ~1.0
    inv = total / (len(counts) * counts)
    weights = {int(k): float(v) for k, v in inv.items()}
    return weights

def build_weighted_sampler(train_df: pd.DataFrame, class_weights: Dict[int, float]) -> WeightedRandomSampler:
    """
    Create a per-sample weight vector mapped from class_weights.
    """
    sample_weights = [class_weights[int(c)] for c in train_df["label_idx"].tolist()]
    sampler = WeightedRandomSampler(weights=sample_weights, num_samples=len(sample_weights), replacement=True)
    return sampler

# --------------------------
# 4) DataLoaders factory
# --------------------------
def build_dataloaders(cfg,
                      use_weighted_sampler: bool = False,
                      persistent_workers: Optional[bool] = None):
    """
    Returns train/val/test DataLoaders with sensible defaults.
    """
    train_csv = cfg.artifacts_dir / "train.csv"
    val_csv   = cfg.artifacts_dir / "val.csv"
    test_csv  = cfg.artifacts_dir / "test.csv"

    train_tfms, eval_tfms = build_transforms(cfg.image_size)

    train_ds = FlowerCsvDataset(train_csv, transform=train_tfms)
    val_ds   = FlowerCsvDataset(val_csv, transform=eval_tfms)
    test_ds  = FlowerCsvDataset(test_csv, transform=eval_tfms)

    # Sampler (optional, helpful for imbalance)
    sampler = None
    if use_weighted_sampler:
        class_w = compute_class_weights(train_csv)
        # Persist for reference
        with open(cfg.artifacts_dir / "class_weights.json", "w") as f:
            json.dump({str(k): v for k, v in class_w.items()}, f, indent=2)
        df_train = pd.read_csv(train_csv)
        sampler = build_weighted_sampler(df_train, class_w)

    # pin_memory helps only with CUDA generally
    pin_mem = (DEVICE.type == "cuda")
    if persistent_workers is None:
        persistent_workers = (cfg.num_workers > 0)

    # Dataloaders
    generator = torch.Generator()
    generator.manual_seed(42)

    train_loader = DataLoader(
        train_ds,
        batch_size=cfg.batch_size,
        shuffle=(sampler is None),
        sampler=sampler,
        num_workers=cfg.num_workers,
        pin_memory=pin_mem,
        drop_last=False,
        worker_init_fn=seed_worker,
        persistent_workers=persistent_workers,
        prefetch_factor=2 if cfg.num_workers > 0 else None,
        generator=generator,
    )

    val_loader = DataLoader(
        val_ds,
        batch_size=max(1, cfg.batch_size // 2),  # eval can be smaller to reduce memory
        shuffle=False,
        num_workers=cfg.num_workers,
        pin_memory=pin_mem,
        drop_last=False,
        worker_init_fn=seed_worker,
        persistent_workers=persistent_workers,
        prefetch_factor=2 if cfg.num_workers > 0 else None,
        generator=generator,
    )
    test_loader = DataLoader(
        test_ds,
        batch_size=max(1, cfg.batch_size // 2),
        shuffle=False,
        num_workers=cfg.num_workers,
        pin_memory=pin_mem,
        drop_last=False,
        worker_init_fn=seed_worker,
        persistent_workers=persistent_workers,
        prefetch_factor=2 if cfg.num_workers > 0 else None,
        generator=generator,
    )

    meta = {
        "num_classes": int(pd.read_csv(train_csv)["label_idx"].nunique()),
        "class_names": sorted(pd.read_csv(cfg.artifacts_dir / "manifest.csv")["label"].unique().tolist()),
    }
    with open(cfg.artifacts_dir / "data_meta.json", "w") as f:
        json.dump(meta, f, indent=2)
    print("Data meta:", meta)

    return train_loader, val_loader, test_loader

# --------------------------
# 5) (Optional) quick preview of an augmented batch
# --------------------------
def show_augmented_batch(train_loader, max_images=12):
    import matplotlib.pyplot as plt
    batch = next(iter(train_loader))
    imgs, targets = batch
    n = min(max_images, imgs.size(0))
    # Unnormalize for display
    mean = torch.tensor(IMAGENET_MEAN).view(3,1,1)
    std  = torch.tensor(IMAGENET_STD).view(3,1,1)

    plt.figure(figsize=(n * 1.7, 1.7 * 2))
    for i in range(n):
        img = imgs[i].cpu() * std + mean
        img = img.clamp(0,1).permute(1,2,0).numpy()
        plt.subplot(2, (n+1)//2, i+1)
        plt.imshow(img)
        plt.title(f"y={int(targets[i])}", fontsize=9)
        plt.axis("off")
    plt.tight_layout()
    plt.show()

# --------------------------
# 6) Run the prep pipeline
# --------------------------
print("Building dataloaders...")
train_loader, val_loader, test_loader = build_dataloaders(CFG, use_weighted_sampler=False)
print("Train/Val/Test sizes:",
      len(train_loader.dataset), len(val_loader.dataset), len(test_loader.dataset))

# Optional: uncomment to visualize augmentations (can be skipped on tight CPU)
# show_augmented_batch(train_loader, max_images=8)
```

Why these choices?

- **Augmentations:** small but effective for natural images; **RandomResizedCrop** handles aspect/scale variation; flips/color/rotation add invariance without heavy compute.
- **ImageNet normalization:** ideal for **transfer learning**; avoids compute to estimate dataset stats.
- **Weighted sampler (optional):** easy switch if class imbalance hurts macro-F1.
- **Worker seeding & persistent workers:** better reproducibility and lighter CPU overhead over epochs.

- **Pin memory:** only when it helps (CUDA); safe no-op elsewhere.

What's next

In **Chunk 4 (Modeling)** we will:

- Initialize a pretrained backbone (**EfficientNet-B0** default; **MobileNetV3-Small** as an even lighter alternative).
- **Freeze** the backbone for a short warmup, then **unfreeze** for fine-tuning.
- Add **regularization**: label smoothing, weight decay, dropout; plus **early stopping** and **OneCycleLR** or cosine schedule.

✅ Quick recap of your requirements (this chunk)

- Provide **modular, well-commented** code to prepare data efficiently.
- Include **augmentations**, CSV-based dataset, **reproducible** DataLoaders, and an **optional imbalance** remedy.
- Keep everything **CPU-friendly** and ready for transfer learning.

🧭 Mindmap update (CRISP-DM)

- **Business Understanding:** ✅
- **Data Understanding:** ✅
- **Data Preparation:** ✅ (this chunk)
- **Modeling:** ⏳ next (transfer learning + regularization + schedules)
- **Evaluation:** later (accuracy, macro-F1, confusion matrix, curves)
- **Deployment:** later (export & inference)

Say **"continue"** when you're ready for **Chunk 4 (Modeling)**.

User:

Continue, Please provide the detailed information, which model is selected and what was the idea behind choosing that model

ChatGPT:

Awesome—here's **Chunk 4** (Modeling). You asked specifically which model we're using and why; I'll explain the choice up front, then give you a clean, modular training harness you can run as-is. When you're ready for the next chunk (Evaluation), say **"continue."**

CHUNK 4 — Modeling (CRISP-DM: Modeling)

Model selection & rationale (resource-efficient transfer learning)

Default: EfficientNet-B0 (ImageNet-pretrained)
Alternative (even lighter): MobileNetV3-Small (ImageNet-pretrained)

Why EfficientNet-B0 by default?

- **Great accuracy per FLOP** on natural images; strong baseline even with few epochs.
- **Moderate size** (~5M parameters; low hundreds of MFLOPs at 224×224), which is still CPU-friendly for a 4k-image dataset.
- **Built-in regularization** (dropout) and scale-aware architecture help generalize with limited data.
- **Stable transfer behavior** with standard ImageNet normalization & common augments.

Why keep MobileNetV3-Small as a switch?

- **Tiny footprint** (~2–3M parameters) and very low compute—handy if you're truly CPU-bound.
- Slightly **lower ceiling** on accuracy vs EfficientNet-B0, but often "good enough" and faster per epoch.

You can toggle in one line via `CFG.model_name`:

- `"efficientnet_b0"` (default) — best balance of accuracy & speed here.
- `"mobilenet_v3_small"` — fastest if you're very constrained.

Modeling code (build model, train with freeze→finetune, early stopping, AMP on CUDA)

> Paste the *entire* cell below into your notebook. It assumes you've run Chunks 1–3 (so `CFG`, dataloaders, and artifacts exist).

```
python
# ==========================
# CRISP-DM: Modeling
# Transfer learning with EfficientNet-B0 (default) or MobileNetV3-Small
# Features: freeze-then-finetune, label smoothing, AdamW, OneCycleLR, early stopping, AMP on CUDA
# ==========================
import contextlib, json, math, time
from dataclasses import asdict
from pathlib import Path
from typing import Tuple, Dict, Any

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import AdamW
from torch.optim.lr_scheduler import OneCycleLR
from torchvision import models

# Reuse globals from previous chunks
assert 'CFG' in globals(), "Missing CFG. Run Chunk 1."
assert 'train_loader' in globals() and 'val_loader' in globals(), "Build dataloaders in Chunk 3 first."

# --------------------------
# 0) Data meta (classes)
# --------------------------
with open(CFG.artifacts_dir / "data_meta.json") as f:
    DATA_META = json.load(f)
NUM_CLASSES = int(DATA_META["num_classes"])
CLASS_NAMES = DATA_META["class_names"]

# --------------------------
# 1) Model builder
# --------------------------
def _get_weights_enum(model_name: str):
    """
    Handle torchvision weights enum across versions.
    Falls back to .DEFAULT if available.
    """
    if model_name == "efficientnet_b0":
        try:
            return models.EfficientNet_B0_Weights.IMAGENET1K_V1
        except AttributeError:
            return models.EfficientNet_B0_Weights.DEFAULT
    elif model_name == "mobilenet_v3_small":
        try:
            return models.MobileNet_V3_Small_Weights.IMAGENET1K_V1
        except AttributeError:
            return models.MobileNet_V3_Small_Weights.DEFAULT
    else:
        raise ValueError(f"Unsupported model_name: {model_name}")

def build_model(model_name: str, num_classes: int) -> nn.Module:
    weights = _get_weights_enum(model_name)
    if model_name == "efficientnet_b0":
        net = models.efficientnet_b0(weights=weights)
        in_features = net.classifier[1].in_features
        # Keep the default dropout in classifier; just swap final linear
        net.classifier[1] = nn.Linear(in_features, num_classes)
    elif model_name == "mobilenet_v3_small":
        net = models.mobilenet_v3_small(weights=weights)
        in_features = net.classifier[-1].in_features
        # MobileNetV3 classifier already has Dropout; replace last Linear
        net.classifier[-1] = nn.Linear(in_features, num_classes)
    else:
        raise ValueError(f"Unsupported model_name: {model_name}")
    return net

def split_param_groups(model: nn.Module, head_key: str = "classifier"):
    """
    Return (backbone_params, head_params) by module name prefix.
    Works for both EfficientNet and MobileNetV3 (they use 'classifier').
    """
    head_params, backbone_params = [], []
    for name, p in model.named_parameters():
        if name.startswith(f"{head_key}."):
            head_params.append(p)
        else:
            backbone_params.append(p)
    return backbone_params, head_params

def set_backbone_trainable(model: nn.Module, trainable: bool, head_key: str = "classifier"):
    for name, p in model.named_parameters():
        if not name.startswith(f"{head_key}."):
            p.requires_grad = trainable

# --------------------------
# 2) Loss function (label smoothing)
# --------------------------
def build_criterion(label_smoothing: float = 0.0):
    try:
        return nn.CrossEntropyLoss(label_smoothing=label_smoothing)
    except TypeError:
        # Fallback if PyTorch is old
        class LSCE(nn.Module):
            def __init__(self, eps):
                super().__init__()
                self.eps = eps
            def forward(self, logits, target):
                logp = F.log_softmax(logits, dim=-1)
                n = logits.size(-1)
                one_hot = torch.zeros_like(logp).scatter(1, target.view(-1,1), 1)
                soft = (1 - self.eps) * one_hot + self.eps / n
                loss = -(soft * logp).sum(dim=-1).mean()
                return loss
        return LSCE(label_smoothing)

# --------------------------
# 3) Metrics
# --------------------------
@torch.no_grad()
def evaluate(model: nn.Module, loader, criterion, device) -> Dict[str, float]:
    model.eval()
    total_loss, total_correct, total_count = 0.0, 0, 0
    for xb, yb in loader:
        xb = xb.to(device, non_blocking=True)
        yb = yb.to(device, non_blocking=True)
        logits = model(xb)
        loss = criterion(logits, yb)
        total_loss += float(loss.item()) * yb.size(0)
        preds = logits.argmax(dim=1)
        total_correct += int((preds == yb).sum().item())
        total_count  += int(yb.size(0))
    return {
        "loss": total_loss / max(1, total_count),
        "acc": total_correct / max(1, total_count),
        "count": total_count,
    }

# --------------------------
# 4) Train loop (freeze → finetune, AMP on CUDA, early stopping, grad clip)
# --------------------------
def fit(cfg, train_loader, val_loader, save_dir: Path):
    device = DEVICE
    model = build_model(cfg.model_name, NUM_CLASSES).to(device)

    # Param groups (smaller LR on backbone)
    backbone_params, head_params = split_param_groups(model, head_key="classifier")
    lr_head = cfg.learning_rate
    lr_backbone = cfg.learning_rate * 0.33
    optimizer = AdamW(
        [
            {"params": backbone_params, "lr": lr_backbone},
            {"params": head_params, "lr": lr_head},
        ],
        weight_decay=cfg.weight_decay,
        betas=(0.9, 0.999),
        eps=1e-8,
    )

    criterion = build_criterion(cfg.label_smoothing)

    # Scheduler: OneCycle across total steps
    steps_per_epoch = max(1, len(train_loader))
    scheduler = OneCycleLR(
        optimizer,
        max_lr=[lr_backbone, lr_head],
        epochs=cfg.epochs,
        steps_per_epoch=steps_per_epoch,
        pct_start=0.15,
        div_factor=10.0,
        final_div_factor=10.0,
        three_phase=False,
        anneal_strategy="cos"
    )

    # AMP on CUDA only (MPS/CPU use full precision)
    use_amp = (cfg.mixed_precision and device.type == "cuda")
    scaler = torch.cuda.amp.GradScaler(enabled=use_amp)
    autocast_ctx = torch.cuda.amp.autocast if use_amp else contextlib.nullcontext

    # Warmup: freeze backbone for first N epochs
    set_backbone_trainable(model, trainable=False)
    frozen_epochs = cfg.freeze_backbone_epochs

    # Early stopping
    best_val_acc = -1.0
    best_state = None
    patience = 3
    bad_epochs = 0

    history = []
    t0 = time.time()

    for epoch in range(cfg.epochs):
        model.train()
        if epoch == frozen_epochs:
            # Unfreeze backbone for fine-tuning
            set_backbone_trainable(model, trainable=True)
        total_loss, total_correct, total_count = 0.0, 0, 0

        for xb, yb in train_loader:
            xb = xb.to(device, non_blocking=True)
            yb = yb.to(device, non_blocking=True)

            optimizer.zero_grad(set_to_none=True)

            with autocast_ctx():
                logits = model(xb)
                loss = criterion(logits, yb)

            if use_amp:
                scaler.scale(loss).backward()
                # Gradient clipping (clip by norm to 1.0)
                scaler.unscale_(optimizer)
                torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
                scaler.step(optimizer)
                scaler.update()
            else:
                loss.backward()
                torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
                optimizer.step()

            scheduler.step()

            total_loss += float(loss.item()) * yb.size(0)
            preds = logits.argmax(dim=1)
            total_correct += int((preds == yb).sum().item())
            total_count  += int(yb.size(0))

        train_loss = total_loss / max(1, total_count)
        train_acc = total_correct / max(1, total_count)

        # Validation
        val_metrics = evaluate(model, val_loader, criterion, device)
        val_loss, val_acc = val_metrics["loss"], val_metrics["acc"]

        epoch_log = {
            "epoch": epoch + 1,
            "train_loss": train_loss, "train_acc": train_acc,
            "val_loss": val_loss, "val_acc": val_acc,
            "frozen_backbone": (epoch < frozen_epochs)
        }
        history.append(epoch_log)
        print(f"[{epoch+1:02d}/{cfg.epochs}] "
              f"train_loss={train_loss:.4f} acc={train_acc:.3f} | "
              f"val_loss={val_loss:.4f} acc={val_acc:.3f} | "
              f"{'FROZEN' if epoch < frozen_epochs else 'FT'}")

        # Early stopping on val_acc
        improved = val_acc > best_val_acc + 1e-4
        if improved:
            best_val_acc = val_acc
            bad_epochs = 0
            best_state = {
                "model_name": cfg.model_name,
                "epoch": epoch + 1,
                "state_dict": model.state_dict(),
                "class_names": CLASS_NAMES,
                "image_size": cfg.image_size,
                "norm_mean": (0.485, 0.456, 0.406),
                "norm_std":  (0.229, 0.224, 0.225),
                "val_acc": best_val_acc,
            }
        else:
            bad_epochs += 1
            if bad_epochs >= patience:
                print(f"Early stopping at epoch {epoch+1} (no val_acc improvement in {patience} epochs).")
                break

    dur = time.time() - t0
    print(f"Training finished in {dur/60:.1f} min. Best val_acc={best_val_acc:.3f}")

    # Persist history and best checkpoint
    save_dir.mkdir(parents=True, exist_ok=True)
    with open(save_dir / "train_history.json", "w") as f:
        json.dump(history, f, indent=2)

    if best_state is not None:
        torch.save(best_state, save_dir / "best_model.pt")
        print(f"Saved best model to {save_dir / 'best_model.pt'} (epoch {best_state['epoch']}).")
    else:
        # Save last state if no improvement recorded (unlikely)
        torch.save({
            "model_name": cfg.model_name,
            "state_dict": model.state_dict(),
            "class_names": CLASS_NAMES,
            "image_size": cfg.image_size,
            "norm_mean": (0.485, 0.456, 0.406),
            "norm_std":  (0.229, 0.224, 0.225),
            "val_acc": None,
        }, save_dir / "best_model.pt")
        print(f"Saved (last) model to {save_dir / 'best_model.pt'}.")

    return best_state, history

# --------------------------
# 5) Run training
# --------------------------
best_state, history = fit(CFG, train_loader, val_loader, CFG.artifacts_dir)
```

Notes on efficiency & best practices

- **Freeze → finetune:** trains the new head quickly, then unfreezes the backbone for a few epochs to refine; this is compute-efficient.
- **Label smoothing + weight decay + dropout:** reduce overfit on small datasets.
- **OneCycleLR:** helps converge in few epochs without extensive tuning.
- **AMP (CUDA only):** speeds up & saves memory if you do have a GPU; safe no-op on CPU/MPS.
- **Early stopping:** avoids wasting epochs once validation stalls.
- **Gradient clipping:** guards against rare exploding gradients when unfreezing.

What's next

In **Chunk 5 (Evaluation & Testing)** we will:

- Load the saved **best checkpoint**.
- Compute **Accuracy** and **Macro-F1** on **val** and **test** sets.
- Show a **confusion matrix** and a **per-class report**.
- Plot **loss/accuracy curves** from `train_history.json`.

✅ Quick recap of your request (this chunk)

- Explain **which model is selected** (**EfficientNet-B0** by default; **MobileNetV3-Small** optional) and **why**.
- Provide **ready-to-run modeling code** with **freeze→finetune**, **regularization**, **scheduler**, **early stopping**, and **AMP** for efficiency.

🧭 Mindmap update (CRISP-DM)

- **Business Understanding:** ✅
- **Data Understanding:** ✅
- **Data Preparation:** ✅
- **Modeling:** ✅ (implemented & trained)
- **Evaluation:** ⏳ next — metrics, confusion matrix, curves
- **Deployment:** later — export + simple inference cell

Say **“continue”** when you want **Chunk 5 (Evaluation & Testing)**.

User:

Lets continue

ChatGPT:

Great—here’s **Chunk 5** (Evaluation & Testing). When you’re ready for the next part, say **“continue.”**

CHUNK 5 — Evaluation & Testing (CRISP-DM: Evaluation)

What we’ll do

- Load the **best checkpoint** (`artifacts/best_model.pt`).
- Rebuild the model (no extra downloads), run **val** and **test** evaluation.
- Report **Accuracy**, **Macro-F1**, **Weighted-F1**, and **Top-3 accuracy**.
- Show **confusion matrices** and a **per-class report**.
- Plot **training curves** from `train_history.json`.
- Save all artifacts to `artifacts/`.

  Paste the entire block below into **one** Jupyter cell.

```
python
# ==========================
# CRISP-DM: Evaluation & Testing
# Load best checkpoint, compute metrics, confusion matrices, and plot training curves
# ==========================
import json, math
from pathlib import Path
import numpy as np
import torch
import torch.nn as nn
from torchvision import models, transforms
from sklearn.metrics import (
    classification_report, confusion_matrix, f1_score,
    accuracy_score, ConfusionMatrixDisplay, top_k_accuracy_score
)
import matplotlib.pyplot as plt
import pandas as pd

assert 'CFG' in globals(), "Missing CFG. Please run Chunk 1."
assert 'val_loader' in globals() and 'test_loader' in globals(), "Missing dataloaders. Please run Chunk 3."
ART = CFG.artifacts_dir
CKPT_PATH = ART / "best_model.pt"
```

```
HIST_PATH = ART / "train_history.json"
assert CKPT_PATH.exists(), f"Checkpoint not found at {CKPT_PATH}. Please run Chunk 4 (training)."

# --------------------------
# 1) Rebuild model skeleton (no pretrained download), then load weights
# --------------------------
def build_skeleton(model_name: str, num_classes: int) -> nn.Module:
    """Create architecture with random init; we'll load trained weights next."""
    if model_name == "efficientnet_b0":
        net = models.efficientnet_b0(weights=None)
        in_features = net.classifier[1].in_features
        net.classifier[1] = nn.Linear(in_features, num_classes)
    elif model_name == "mobilenet_v3_small":
        net = models.mobilenet_v3_small(weights=None)
        in_features = net.classifier[-1].in_features
        net.classifier[-1] = nn.Linear(in_features, num_classes)
    else:
        raise ValueError(f"Unsupported model_name: {model_name}")
    return net

ckpt = torch.load(CKPT_PATH, map_location=DEVICE)
model_name = ckpt.get("model_name", CFG.model_name)
class_names = ckpt.get("class_names")
num_classes = len(class_names)
image_size = ckpt.get("image_size", CFG.image_size)
norm_mean = ckpt.get("norm_mean", (0.485,0.456,0.406))
norm_std  = ckpt.get("norm_std",  (0.229,0.224,0.225))

model = build_skeleton(model_name, num_classes).to(DEVICE)
model.load_state_dict(ckpt["state_dict"], strict=True)
model.eval()

# --------------------------
# 2) Helper: run inference over a loader
# --------------------------
@torch.no_grad()
def predict_loader(model, loader, device=DEVICE):
    y_true, y_pred, y_scores = [], [], []
    for xb, yb in loader:
        xb = xb.to(device, non_blocking=True)
        logits = model(xb)
        probs = torch.softmax(logits, dim=1)
        preds = probs.argmax(1)
        y_true.append(yb.cpu().numpy())
        y_pred.append(preds.cpu().numpy())
        y_scores.append(probs.cpu().numpy())
    y_true = np.concatenate(y_true)
    y_pred = np.concatenate(y_pred)
    y_scores = np.concatenate(y_scores)
    return y_true, y_pred, y_scores

def eval_and_report(split_name, loader):
    y_true, y_pred, y_scores = predict_loader(model, loader, DEVICE)

    acc = accuracy_score(y_true, y_pred)
    macro_f1 = f1_score(y_true, y_pred, average="macro")
    weighted_f1 = f1_score(y_true, y_pred, average="weighted")

    # Top-3 accuracy (robust to class confusions)
    try:
        top3 = top_k_accuracy_score(y_true, y_scores, k=min(3, y_scores.shape[1]))
    except Exception:
        top3 = None

    # Per-class report
    report_dict = classification_report(
        y_true, y_pred, target_names=class_names, output_dict=True, zero_division=0
    )
    report_df = pd.DataFrame(report_dict).transpose()
    report_csv = ART / f"classification_report_{split_name}.csv"
    report_df.to_csv(report_csv, index=True)

    # Confusion matrices (raw and normalized)
    cm = confusion_matrix(y_true, y_pred, labels=list(range(len(class_names))))
    cm_norm = confusion_matrix(y_true, y_pred, labels=list(range(len(class_names))), normalize="true")

    fig1, ax1 = plt.subplots(figsize=(6,5))
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
    disp.plot(include_values=True, cmap=None, ax=ax1, xticks_rotation=45, colorbar=False)
    plt.title(f"{split_name} Confusion Matrix (counts)")
    plt.tight_layout()
    cm_counts_png = ART / f"confusion_matrix_{split_name}_counts.png"
    plt.savefig(cm_counts_png, dpi=160)
    plt.show()

    fig2, ax2 = plt.subplots(figsize=(6,5))
    disp2 = ConfusionMatrixDisplay(confusion_matrix=cm_norm, display_labels=class_names)
    disp2.plot(include_values=True, cmap=None, ax=ax2, xticks_rotation=45, colorbar=False)
    plt.title(f"{split_name} Confusion Matrix (row-normalized)")
    plt.tight_layout()
    cm_norm_png = ART / f"confusion_matrix_{split_name}_normalized.png"
    plt.savefig(cm_norm_png, dpi=160)
    plt.show()

    # Save summary JSON
    summary = {
        "split": split_name,
        "accuracy": float(acc),
        "macro_f1": float(macro_f1),
        "weighted_f1": float(weighted_f1),
        "top3_accuracy": (float(top3) if top3 is not None else None),
        "num_samples": int(len(y_true)),
        "classes": class_names,
        "report_csv": str(report_csv),
        "confusion_counts_png": str(cm_counts_png),
        "confusion_norm_png": str(cm_norm_png),
    }
    with open(ART / f"eval_{split_name}.json", "w") as f:
        json.dump(summary, f, indent=2)

    # Print concise summary
    print(f"\n== {split_name.upper()} RESULTS ==")
    print(f"Accuracy:     {acc:.4f}")
    print(f"Macro-F1:     {macro_f1:.4f}")
    print(f"Weighted-F1:  {weighted_f1:.4f}")
    if top3 is not None:
        print(f"Top-3 Acc:    {top3:.4f}")
    print(f"Per-class report saved to: {report_csv}")
    print(f"Confusion matrices saved to:\n  {cm_counts_png}\n  {cm_norm_png}")

    return summary, (y_true, y_pred, y_scores)

val_summary, (y_true_val, y_pred_val, y_scores_val) = eval_and_report("val", val_loader)
test_summary, (y_true_test, y_pred_test, y_scores_test) = eval_and_report("test", test_loader)

# --------------------------
# 3) Plot training curves from history
# --------------------------
if HIST_PATH.exists():
    with open(HIST_PATH) as f:
        hist = json.load(f)
    # Convert to DataFrame for convenience
    hist_df = pd.DataFrame(hist)
    # Curves
    plt.figure(figsize=(6,4))
    plt.plot(hist_df["epoch"], hist_df["train_loss"], label="train_loss")
    plt.plot(hist_df["epoch"], hist_df["val_loss"], label="val_loss")
    plt.xlabel("Epoch"); plt.ylabel("Loss"); plt.title("Training vs Validation Loss")
    plt.legend(); plt.tight_layout()
    loss_png = ART / "curve_loss.png"
    plt.savefig(loss_png, dpi=160)
    plt.show()

    plt.figure(figsize=(6,4))
    plt.plot(hist_df["epoch"], hist_df["train_acc"], label="train_acc")
    plt.plot(hist_df["epoch"], hist_df["val_acc"], label="val_acc")
    plt.xlabel("Epoch"); plt.ylabel("Accuracy"); plt.title("Training vs Validation Accuracy")
    plt.legend(); plt.tight_layout()
    acc_png = ART / "curve_accuracy.png"
    plt.savefig(acc_png, dpi=160)
    plt.show()

    print(f"Saved curves:\n  {loss_png}\n  {acc_png}")
else:
    print(f"No history file at {HIST_PATH}; skipping curves.")

# --------------------------
# 4) (Optional) Most-confused pairs helper (simple, fast)
# --------------------------
def most_confused_pairs(y_true, y_pred, class_names, top_k=5):
    cm = confusion_matrix(y_true, y_pred, labels=list(range(len(class_names))))
    np.fill_diagonal(cm, 0)  # ignore correct predictions
    pairs = []
    for i in range(len(class_names)):
        for j in range(len(class_names)):
            if i != j and cm[i, j] > 0:
                pairs.append((class_names[i], class_names[j], int(cm[i, j])))
    pairs.sort(key=lambda x: x[2], reverse=True)
    return pairs[:top_k]

pairs = most_confused_pairs(y_true_val, y_pred_val, class_names, top_k=5)
if pairs:
    print("\nMost-confused label pairs on VAL:")
    for a, b, c in pairs:
        print(f"  {a} → {b}: {c} samples")
```

Reading the results

- Check the printed **val/test summaries** for the core metrics.
- Open the saved CSVs for detailed **per-class precision/recall/F1**.
- Inspect both **confusion matrices** (counts & row-normalized) to see where misclassifications happen.
- Look at **training curves** to diagnose under/overfitting (e.g., rising val loss).

Tips if accuracy plateaus

- Toggle to **MobileNetV3-Small** only if you need more speed; otherwise keep **EfficientNet-B0**.
- Slightly increase **image_size** (256) if you have headroom.
- Enable **WeightedRandomSampler** (set `use_weighted_sampler=True` in Chunk 3) if imbalance hurts Mac-F1.
- Add mild **CutMix/MixUp** (optional) or train a few more epochs with a **lower LR** and unfreezed backbone.

✅ Quick recap of your requirements (this chunk)

- Provide code to **evaluate** the trained model: Accuracy, Macro-F1, Weighted-F1, **Top-3**, confusion matrices, per-class report, and training **curves**—all **resource-efficient** and saved to `artifacts/`.

🧭 Mindmap update (CRISP-DM)

- **Business Understanding:** ✅
- **Data Understanding:** ✅
- **Data Preparation:** ✅
- **Modeling:** ✅
- **Evaluation:** ✅ (this chunk)
- **Deployment:** ⏳ next — lightweight export & an inference helper cell

Say **“continue”** when you’re ready for **Chunk 6 (Deployment & Inference)**.

User:

explain the results == VAL RESULTS == Accuracy: 0.9414 Macro-F1: 0.9404 Weighted-F1: 0.9414 Top-3 Acc: 0.9923

ChatGPT:

Here’s how to read those **validation results** at a glance:

- **Top-1 Accuracy = 0.9414 (94.14%)**
  ~“Out of ~every 100 images, ≈94 are correctly classified.” If your val split is ~15% of 4k (≈600 images), that’s roughly **~35 misclassifications**. A rough 95% binomial CI would be about **±2%** (so ≈92–96%), assuming ~600 val samples.
- **Macro-F1 = 0.9404**
  Averages F1 **equally across classes**. Being within ~0.001 of accuracy suggests **consistently strong performance** across all flower types (no class is dragging you down badly).
- **Weighted-F1 = 0.9414**
  Weights each class by its support. The fact that it **matches accuracy** hints that your class distribution isn’t severely imbalanced, and precision/recall are both healthy.
- **Top-3 Accuracy = 0.9923 (99.23%)**
  The correct class is almost always **among the model’s top 3 guesses**. This means the model’s **ranking confidence is excellent**; remaining errors are typically **close calls between visually similar classes** (e.g., daisy ↔ dandelion, tulip ↔ rose buds).

What this tells us (CRISP-DM: Evaluation insights)

1. **Generalization looks strong.** Freeze→finetune + light augments seem sufficient for this dataset size.
2. **No red flags for imbalance.** Macro-F1 ≈ Weighted-F1 ≈ Accuracy suggests balanced performance across classes.
3. **Residual errors are “near-misses.”** The big Top-3 vs Top-1 gap (~5 percentage points) implies **fine-grained confusions** rather than systematic failures.

Quick checks to deepen the evaluation

- **Per-class report & confusion matrices** (already saved in `artifacts/`):
  Look for pairs with systematic confusion. If you used the helper, check the printed “**Most-confused label pairs**” on VAL to target fixes.
- **Calibration (optional):** If you need well-calibrated probabilities, consider **temperature scaling** on the val set (cheap to run).

Low-cost ways to squeeze a bit more accuracy (if you want)

- **Test-Time Augmentation (TTA):** average predictions over a few flips/crops; cheap bump on CPU.
- **+2–3 epochs with a lower max LR** (keep early stopping): may refine the unfreezed backbone slightly.
- **Image size 256** (only if you have a bit of headroom): small but sometimes meaningful gains.
- **Weighted sampler** (if per-class F1 shows any tail weakness): set `use_weighted_sampler=True` in Chunk 3.

✅ Quick recap of your request (this chunk)

- **Explain the results:** what each metric means, what they imply about balance and error types, and **actionable** next steps to improve or verify.

🧭 Mindmap update (CRISP-DM)

- **Business Understanding:** ✅
- **Data Understanding:** ✅
- **Data Preparation:** ✅
- **Modeling:** ✅
- **Evaluation:** ✅ (interpreted the metrics; inspect confusion matrices next if needed)
- **Deployment:** ⏳ up next — export the best model and provide a lightweight inference cell.

Say **"continue"** if you'd like me to deliver **Chunk 6 (Deployment & Inference)**.

User:

Lets continue and finish the model

ChatGPT:

Awesome—let's wrap with **Chunk 6** (Deployment & Inference). This finishes the end-to-end CRISP-DM flow for your classifier. When you want any optional polishing (e.g., TTA out of the box, ONNX, or lightweight error analysis), say **"continue."**

CHUNK 6 — Deployment & Inference (CRISP-DM: Deployment)

What we'll do

- Load the **best checkpoint** and rebuild the model for inference.
- Provide **single-image** and **batch** prediction helpers with **top-k** outputs.
- Add optional **Test-Time Augmentation (TTA)** (flip) that's CPU-friendly.
- Export options: **TorchScript** (portable `.pt`) and **ONNX** (optional).
- Save the **label mapping** and a tiny **inference demo** against a few test images.

  Paste the entire block below into **one** Jupyter cell.

```
python
# ==========================
# CRISP-DM: Deployment & Inference
# Load best checkpoint; single/batch inference; optional TTA; export (TorchScript/ONNX)
# ==========================
from pathlib import Path
import json, math, random
from typing import List, Dict, Tuple
import numpy as np
from PIL import Image

import torch
import torch.nn as nn
from torchvision import models, transforms

assert 'CFG' in globals(), "CFG not found. Run Chunk 1."
ART = CFG.artifacts_dir
CKPT_PATH = ART / "best_model.pt"
assert CKPT_PATH.exists(), f"Missing {CKPT_PATH}. Train the model first (Chunk 4)."
```

```
# --------- Load checkpoint metadata ---------
ckpt = torch.load(CKPT_PATH, map_location=DEVICE)
MODEL_NAME  = ckpt.get("model_name", CFG.model_name)
CLASS_NAMES = ckpt["class_names"]
NUM_CLASSES = len(CLASS_NAMES)
IMG_SIZE    = ckpt.get("image_size", CFG.image_size)
NORM_MEAN   = ckpt.get("norm_mean", (0.485, 0.456, 0.406))
NORM_STD    = ckpt.get("norm_std",  (0.229, 0.224, 0.225))

# Persist label mapping for external consumers
labels_txt = ART / "labels.txt"
with open(labels_txt, "w") as f:
    for cls in CLASS_NAMES:
        f.write(cls + "\n")
print(f"Saved label mapping to {labels_txt}")

# --------- Transforms for inference (match evaluation) ---------
def infer_transforms(image_size: int = IMG_SIZE):
    return transforms.Compose([
        transforms.Resize(int(image_size * 1.15)),
        transforms.CenterCrop(image_size),
        transforms.ToTensor(),
        transforms.Normalize(NORM_MEAN, NORM_STD),
    ])

TFM_INFER = infer_transforms(IMG_SIZE)

# --------- Build skeleton & load weights (no extra download) ---------
def build_skeleton(model_name: str, num_classes: int) -> nn.Module:
    if model_name == "efficientnet_b0":
        net = models.efficientnet_b0(weights=None)
        in_features = net.classifier[1].in_features
        net.classifier[1] = nn.Linear(in_features, num_classes)
    elif model_name == "mobilenet_v3_small":
        net = models.mobilenet_v3_small(weights=None)
        in_features = net.classifier[-1].in_features
        net.classifier[-1] = nn.Linear(in_features, num_classes)
    else:
        raise ValueError(f"Unsupported model_name: {model_name}")
    return net

MODEL = build_skeleton(MODEL_NAME, NUM_CLASSES).to(DEVICE)
MODEL.load_state_dict(ckpt["state_dict"], strict=True)
MODEL.eval()
print(f"Loaded model '{MODEL_NAME}' with {NUM_CLASSES} classes on device={DEVICE}.")

# --------- Helpers ---------
def _load_image(path: str) -> Image.Image:
    img = Image.open(path).convert("RGB")
    return img

@torch.no_grad()
def predict_tensor(t: torch.Tensor, model: nn.Module, topk: int = 3) -> Dict:
    """
    t: Tensor of shape [C,H,W] (single image, normalized).
    Returns dict with topk indices, labels, probs.
    """
    model.eval()
    xb = t.unsqueeze(0).to(DEVICE, non_blocking=True)  # [1, C, H, W]
    logits = model(xb)
    probs = torch.softmax(logits, dim=1).squeeze(0)    # [C]
    topk = min(topk, probs.shape[0])
    vals, idxs = torch.topk(probs, k=topk)
    idxs = idxs.cpu().tolist()
    vals = vals.cpu().tolist()
    return {
        "topk_indices": idxs,
        "topk_labels": [CLASS_NAMES[i] for i in idxs],
        "topk_probs": [float(p) for p in vals],
    }

@torch.no_grad()
def predict_image(path: str, model: nn.Module = MODEL, topk: int = 3, tta: bool = False) -> Dict:
    """
    Runs inference on a single image file. If tta=True, averages original + hflip.
    """
    img = _load_image(path)
    t = TFM_INFER(img)
    if not tta:
        res = predict_tensor(t, model, topk=topk)
        res["path"] = path
        return res
    else:
        # Simple TTA: original + horizontal flip (very cheap)
        t_flip = transforms.functional.hflip(img)
        t1 = TFM_INFER(img).unsqueeze(0)
        t2 = TFM_INFER(t_flip).unsqueeze(0)
        xb = torch.cat([t1, t2], dim=0).to(DEVICE, non_blocking=True)  # [2, C, H, W]
        logits = model(xb).mean(dim=0, keepdim=True)                   # average logits
        probs = torch.softmax(logits, dim=1).squeeze(0)
        k = min(topk, probs.shape[0])
        vals, idxs = torch.topk(probs, k=k)
        idxs = idxs.cpu().tolist(); vals = vals.cpu().tolist()
        return {
            "path": path,
            "topk_indices": idxs,
            "topk_labels": [CLASS_NAMES[i] for i in idxs],
            "topk_probs": [float(p) for p in vals],
            "tta": "hflip",
        }

@torch.no_grad()
def predict_batch(paths: List[str], model: nn.Module = MODEL, topk: int = 3, tta: bool = False) -> List[Dict]:
    results = []
    for p in paths:
        try:
            results.append(predict_image(p, model, topk=topk, tta=tta))
        except Exception as e:
            results.append({"path": p, "error": str(e)})
    return results

def pretty_print_prediction(pred: Dict):
    if "error" in pred:
        print(f"[ERROR] {pred['path']}: {pred['error']}")
        return
    print(f"\nFile: {pred['path']}")
    for i, (lbl, prob) in enumerate(zip(pred["topk_labels"], pred["topk_probs"]), 1):
        print(f"  {i}. {lbl:>12s}  p={prob:.3f}")

# --------- Quick demo on a few test images (optional) ---------
test_csv = ART / "test.csv"
if test_csv.exists():
    import pandas as pd
    test_df = pd.read_csv(test_csv)
    demo_paths = test_df.sample(n=min(5, len(test_df)), random_state=123)["path"].tolist()
    print("\nDemo predictions on a few test images:")
    preds = predict_batch(demo_paths, MODEL, topk=3, tta=False)
    for p in preds:
        pretty_print_prediction(p)
else:
    print("No test.csv found for demo. Skipping.")

# --------- Export: TorchScript (portable .pt) ---------
def export_torchscript(model: nn.Module, save_path: Path, image_size: int = IMG_SIZE):
    model = model.to("cpu").eval()  # portable
    dummy = torch.randn(1, 3, image_size, image_size)
    with torch.no_grad():
        traced = torch.jit.trace(model, dummy)
    torch.jit.save(traced, save_path)
    print(f"TorchScript saved to: {save_path}")

# Uncomment to export TorchScript
# export_torchscript(MODEL, ART / "model_torchscript.pt", image_size=IMG_SIZE)

# --------- Export: ONNX (optional) ---------
def export_onnx(model: nn.Module, save_path: Path, image_size: int = IMG_SIZE):
    try:
        import onnx  # noqa: F401
    except ImportError:
        print("onnx not installed; run `pip install onnx onnxruntime` to enable ONNX export.")
        return
    model = model.to("cpu").eval()
    dummy = torch.randn(1, 3, image_size, image_size)
    torch.onnx.export(
        model, dummy, str(save_path),
        input_names=["input"], output_names=["logits"],
        opset_version=12, dynamic_axes={"input": {0: "batch"}, "logits": {0: "batch"}}
    )
    print(f"ONNX saved to: {save_path}")

# Uncomment to export ONNX
# export_onnx(MODEL, ART / "model.onnx", image_size=IMG_SIZE)
```

Usage cheatsheet

- **Single image:**

  ```
  python
  pred = predict_image("path/to/your_image.jpg", topk=3, tta=False)
  pretty_print_prediction(pred)
  ```

- **Batch of images:**

  ```
  python
  preds = predict_batch(["img1.jpg", "img2.jpg"], topk=5, tta=True)
  for p in preds: pretty_print_prediction(p)
  ```

- **Export (portable):**

  ```
  python
  export_torchscript(MODEL, CFG.artifacts_dir / "model_torchscript.pt")
  # or
  export_onnx(MODEL, CFG.artifacts_dir / "model.onnx")
  ```

Why these deployment choices (efficient & practical)

- **Same preprocessing** as validation ensures alignment (Resize→CenterCrop→Normalize).
- **TorchScript** gives you a self-contained artifact that runs on CPU without Python model code.
- **ONNX** is optional if you want to serve via **onnxruntime** or move to other runtimes.
- **TTA** is cheap (1 flip) and often nets a tiny bump without heavy compute.

✅ Quick recap of your requirements (this chunk)

- Provide a **ready-to-run inference module**, consistent preprocessing, **top-k outputs**, **batch support**, and **portable exports**—all optimized for **resource-constrained** environments.

🧭 Mindmap update (CRISP-DM)

- **Business Understanding:** ✅
- **Data Understanding:** ✅
- **Data Preparation:** ✅
- **Modeling:** ✅
- **Evaluation:** ✅
- **Deployment:** ✅ (inference + export complete)

If you want optional **Chunk 7 (Error Analysis & Iteration Plan)**—e.g., per-class improvements, lightweight augment tweaks, or simple dataset audits—say **“continue.”**