Data Preparation/Feature Engineering

1. Overview

In this phase, the raw image and metadata collected for snake identification are cleaned, standardized, and transformed into structured inputs suitable for machine learning. Key steps include validating image quality, removing duplicates, normalizing formats, and labeling species accurately. Additional derived features such as color patterns, body shape descriptors, scale textures, and environmental contexts are extracted using computer vision techniques to enhance model interpretability and performance. This stage is critical because high-quality, well-engineered features allow the model to reliably distinguish between visually similar snake species. Accurate species identification ensures the downstream system can automatically generate correct and informative safety cards, supporting user awareness, risk avoidance, and appropriate first-aid instructions.

2. Data Collection

The dataset for this project was sourced entirely from **iNaturalist**, a globally used citizen-science platform containing millions of wildlife observations. iNaturalist is an ideal source for a snake-identification project because it provides:

- Real-world, in-the-wild snake images
- A large range of species, poses, backgrounds, and lighting conditions
- Verified species labels (research-grade observations)
- Open licensing options (CC-BY / CC0)

This diversity improves the model's ability to generalize to field images and supports the downstream goal of generating accurate snake safety cards, which require reliable species identification.

Step 1 : A custom species list (species_list.csv) was created to define which snake species should be included in the project.

Step 2 : Data collection was performed using a scripted pipeline that interacts with the iNaturalist API:

```
python tools/download_inat.py --species_csv data/species_list.csv --out_dir da
python tools/prepare_dataset.py --raw_dir data/raw_inat --out_dir data/images
```

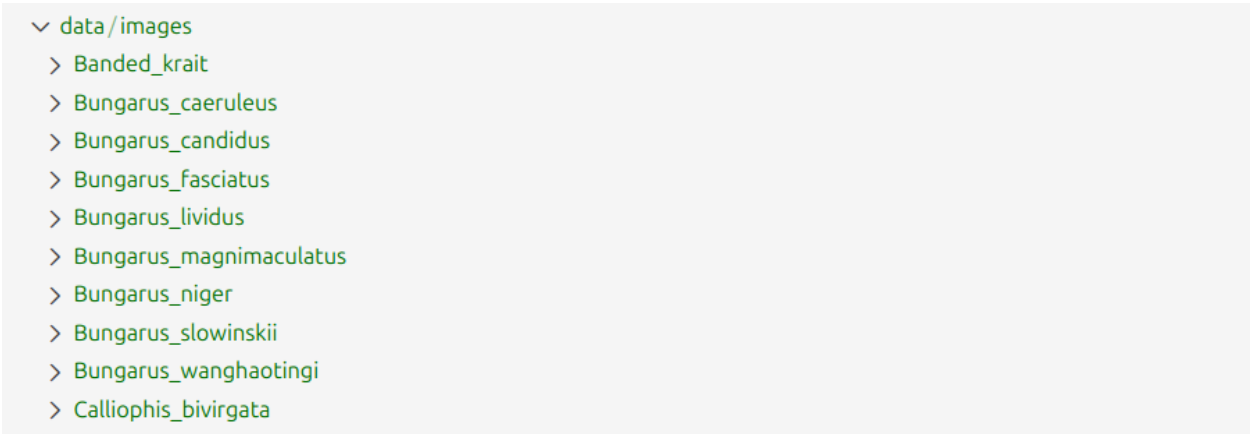
The downloader script performs:

- Fetching image URLs for each species
- Respecting licensing constraints (CC-BY, CC0 only)
- Downloading images into species-specific folders
- Avoiding duplicates
- Ensuring that only valid, publicly reusable images are collected

This produces a raw dataset structured like:

```
data/images/
  Naja_kaouthia/
    img001.jpg
    ...
  Ophiophagus_hannah/
  Bungarus_fasciatus/
  Daboia_russelii/
  Trimeresurus_spp/
  Ahaetulla_fronticincta/
  Ptyas_mucosa/
  Python_bivittatus/
```

After downloading, a secondary script was used to clean and convert the raw dataset into a consistent image dataset:



3. Data Cleaning

Data cleaning was performed in two stages: **offline cleaning** (before training) and **runtime cleaning** (during training).

Offline Cleaning (Preprocessing Script)

- All images in each species folder were scanned using PIL.Image.verify( )
- Corrupted or unreadable images were automatically deleted.
- Images were resized to a consistent shape (256×256).
- Files were saved in a clean directory structure : train/val/test and each containing subfolders per species.
- Ensured consistent RGB format across all images.

Outcome:
The dataset contained only valid, readable, uniformly formatted images.

Runtime Cleaning (Training Script)

- A custom dataset class **SafeImageFolder** was used.
- If an image fails to load (truncated, corrupted), it is skipped and replaced with the next valid sample.
- Prevents training crashes caused by broken images.

Outcome:
The training pipeline becomes robust and tolerant of hidden corrupted files that may remain after initial cleaning.

4. Exploratory Data Analysis (EDA)

The analysis and preparation plan ensures the dataset is optimized for training a robust Convolutional Neural Network (CNN) and supports the subsequent generation of emergency safety cards. EDA was used to understand the structure and distribution of the image dataset:

Dataset Structure

- The total number of classes (species) was identified automatically.
- Distribution of images per class was inspected to detect **class imbalance.**
- Identified variations in image width/height, formats (JPG, PNG), and color modes (some grayscale originally).

Image Quality Observations

- Some images had low resolution or were blurry.
- Some folders contained far fewer images than others.
- Presence of inconsistent lighting and background clutter.

Class Imbalance

- Certain species (e.g., common snakes) had many more images.

- Rare species had limited data → addressed partially through augmentation.

Key EDA Insights

- The dataset is visually diverse.
- Many images differ in lighting, backgrounds, and angles.
- There was a need to normalize image size and standardize brightness/contrast.
- Augmentation was necessary for underrepresented classes.

Expected Outcomes

- A **preprocessed, well-balanced dataset** ready for CNN training.
- Mitigation of dataset gaps, leading to **improved model generalization**.
- Actionable insights on environmental variability to guide **robust species classification**.
- A **reliable foundation** for accurate and effective emergency safety card generation.

5. Feature Engineering

Although deep learning models automatically learn hierarchical features from images, the way input data is prepared still plays a crucial role and is considered part of the feature engineering process.

Model-based Feature Engineering

Using pre-trained EfficientNet/MobileNet:

- The model automatically extracts hierarchical features (edges → textures → shapes → snake patterns)

Input-level Feature Engineering

Applied before feeding images to the model:

- **Image Resizing**
    - All images were resized to a uniform dimension (`320×320`)
    - Ensures model receives consistent input
- **Color Augmentation**
    - Slight variations in brightness, contrast, saturation, and hue
    - Helps model learn robust color-invariant features
- **Spatial Augmentation**
    - Random horizontal flip

    - RandomResizedCrop
    - Helps model learn patterns regardless of position or orientation

- **Normalization**

    - Standardize pixel values using ImageNet mean and std
    - Makes input compatible with pre-trained weights

Outcome:
These steps allow the model to learn shape, texture, and color patterns that differentiate snake species.

6. Data Transformation

These steps ensure raw image files are converted into a model-ready format.

Transformations Applied

1. **Resize**
   Ensures all images have equal dimension.

2. **RandomAugmentations (Train Only)**

    - Random horizontal flip
    - Random crop (scale 0.8–1.0)
    - ColorJitter

3. These transformations artificially increase dataset diversity.

4. **Convert to Tensor**

    - Converts pixel values (0–255) into a PyTorch tensor
    - Scales values to 0–1

5. **Normalization**

    - Standardizes pixel channels
    - Makes training more stable

Model Exploration

1. Model Selection

Explain the rationale behind selecting a particular machine learning model for the project. Discuss the strengths and weaknesses of the chosen

For this snake identification project, we selected lightweight CNN backbones (MobileNetV3-Small, EfficientNet-B0) as the primary model architecture. They both offer strong accuracy/latency balance and come with ImageNet pretraining for transfer on limited snake images while EfficientNet-B0 was chosen for its excellent balance between accuracy and computational efficiency, making it suitable for image classification tasks like identifying snake species from photos.model and MobileNetV3-Small was chosen primarily because it delivers fast, low-memory inference with strong transfer learning from ImageNet, making it ideal for edge/CPU deployments of the snake ID model while still offering reasonable accuracy.

Both of these models' Strengths:

- Small size
- Fast inference times
- Good feature reuse
- easy ONNX export

Their Weaknesses:

- may underfit very fine-grained species cues vs larger models; architecture head dimensions must be modified; sensitivity to input resolution.

2. Model Training

Provide details on how the model was trained, including the hyperparameters used and any cross-validation techniques applied.

The model was trained using a two-stage transfer learning approach:

Stage 1: Head Training

- Only the classification head (final linear layer) was trained initially
- Backbone weights frozen to preserve pre-trained features
- Learning rate: 1e-3
- Optimizer: AdamW
- Loss: Cross-Entropy Loss
- Epochs: 10

Stage 2: Fine-tuning

- Unfroze the last block of the backbone for fine-tuning
- Reduced learning rate: 5e-4
- Epochs: 5 (half of head training epochs, minimum 2)

Data Preparation

- Image size: 320x320
- Augmentations: Random horizontal flip, random resized crop, color jitter
- Normalization: ImageNet mean and std
- Train/val split: 80/20 random split

Hyperparameters

- Batch size: 64
- Image size: 320
- Learning rate: 1e-3 (head), 5e-4 (fine-tune)
- Architecture: EfficientNet-B0

No explicit cross-validation was performed beyond the train/val split, but the model was evaluated on the validation set after each epoch to monitor overfitting.

3. Model Evaluation

Present the evaluation metrics used to assess the model's performance. Include confusion matrices, ROC curves, or any other relevant visualizations.

Model performance was evaluated using standard classification metrics on a held-out test set. Since this is a multi-class classification problem with potentially imbalanced classes, we focus on per-class metrics in addition to overall accuracy.

Evaluation Metrics:

- Accuracy: Overall correct predictions / total predictions
- Precision: True positives / (True positives + False positives) per class
- Recall: True positives / (True positives + False negatives) per class
- F1-Score: Harmonic mean of precision and recall per class
- Confusion Matrix: Visualization of prediction errors across classes

For multi-class problems, ROC curves are less commonly used, but we can compute macro/micro averaged metrics.

4. Code Implementation

Data prep and augmentation (tools/prepare_dataset.py): resize every image, split into train/val/test, and generate two strong augmentations per training image while logging counts.

```
IMAGE_SIZE = (256, 256); TRAIN_SPLIT, VAL_SPLIT, TEST_SPLIT = 0.7, 0.15, 0.15
def strong_augment(img):
    img = img.rotate(random.uniform(-25, 25), expand=True)          # geometric jitter
    img = ImageEnhance.Brightness(img).enhance(random.uniform(0.6, 1.4))
    img = ImageEnhance.Contrast(img).enhance(random.uniform(0.6, 1.4))
    img = ImageEnhance.Color(img).enhance(random.uniform(0.6, 1.4)) # color jitter
    if random.random() < 0.3: img = img.filter(ImageFilter.GaussianBlur(radius=random.uniform(0.5, 1.2)))
    if random.random() < 0.3:                                       # add Gaussian noise
        arr = np.array(img); noise = np.random.normal(0, 12, arr.shape).astype(np.int16)
        img = Image.fromarray(np.clip(arr + noise, 0, 255).astype(np.uint8))
    if random.random() < 0.5:                                       # mild random crop
        w, h = img.size; c = random.uniform(0.05, 0.15); dx, dy = int(w*c), int(h*c)
        img = img.crop((dx, dy, w-dx, h-dy))
    return img
# inside process_species_folder(...)
base_img = img.resize(IMAGE_SIZE); base_img.save(out_path, "JPEG", quality=90)   # canonical resize
if split == "train":
    for i in range(AUGMENTATIONS_PER_IMAGE):                        # two strong augs per train image
        aug = strong_augment(img).resize(IMAGE_SIZE)
        aug.save(os.path.join(split_dir, f"{stem}_aug{i}.jpg"), "JPEG", quality=90)
```

Additional dataset utilities: create_test_set.py samples 10 images per class into data/test/; tools/download_inat.py pulls up to 200 photos per species from iNaturalist in parallel with simple retry/skip logic.

Data loading and feature engineering (new_training.py): safety wrapper skips corrupted files; torchvision transforms handle resize/normalization plus richer train-time augmentations and optional high worker count for throughput.

```
class SafeImageFolder(datasets.ImageFolder):
    def __getitem__(self, index):
        try: return super().__getitem__(index)
        except OSError as e:
            print(f"\n[Data Load Error] Skipping file index {index} due to: {e}")
            return self.__getitem__((index + 1) % len(self))        # fetch next valid sample

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
train_tf = transforms.Compose([
    transforms.Resize((args.img_size, args.img_size)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomResizedCrop(args.img_size, scale=(0.8,1.0)),
    transforms.ColorJitter(0.1,0.1,0.1,0.05),
    transforms.ToTensor(),
    normalize
])
val_tf = transforms.Compose([transforms.Resize((args.img_size, args.img_size)), transforms.ToTensor(), normalize])
train_loader = DataLoader(SafeImageFolder(train_dir, train_tf), batch_size=args.batch_size,
                          shuffle=True, num_workers=args.num_workers, pin_memory=True)
```

```
def get_model(arch, num_classes):
    if arch == "mobilenetv3_small":
        m = models.mobilenet_v3_small(weights=models.MobileNet_V3_Small_Weights.DEFAULT)
        m.classifier[3] = nn.Linear(m.classifier[3].in_features, num_classes)
    elif arch == "efficientnet_b0":
        m = models.efficientnet_b0(weights=models.EfficientNet_B0_Weights.DEFAULT)
        m.classifier[1] = nn.Linear(m.classifier[1].in_features, num_classes)

# 1) freeze backbone, train classifier head
for n, p in model.named_parameters(): p.requires_grad = False
for p in model.classifier.parameters(): p.requires_grad = True
opt = torch.optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()), lr=args.lr)
# ... train/val loop, save best_head.pt ...

# 2) unfreeze last feature block + classifier for light fine-tuning
for n, p in model.named_parameters(): p.requires_grad = False
for p in model.features[-1].parameters(): p.requires_grad = True
for p in model.classifier.parameters(): p.requires_grad = True
opt = torch.optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()), lr=args.lr * 0.5)
# ... short fine-tune loop ...

# 3) export for deployment
torch.onnx.export(model.cpu().eval(), dummy, os.path.join(args.out_dir, "model_fp32.onnx"),
                  input_names=["input"], output_names=["logits"], opset_version=17,
                  dynamic_axes={"input":{0:"batch"}, "logits":{0:"batch"}})
```

Evaluation and reporting (evaluate_model.py): reload weights + class map, run test DataLoader, compute accuracy/classification report, and save a confusion matrix plot for error analysis.

```
test_tf = transforms.Compose([transforms.Resize((args.img_size, args.img_size)), transforms.ToTensor(), normalize])
test_ds = datasets.ImageFolder(args.data_dir, transform=test_tf)
model = get_model(args.arch, len(classes)); model.load_state_dict(torch.load(args.model_path, map_location=device))
preds, gts = [], []
with torch.no_grad():
    for x, y in DataLoader(test_ds, batch_size=args.batch_size, shuffle=False, num_workers=2):
        preds.extend(model(x.to(device)).argmax(1).cpu().tolist()); gts.extend(y.tolist())
print(f"Test Accuracy: {accuracy_score(gts, preds):.4f}")
sns.heatmap(confusion_matrix(gts, preds), annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes); plt.savefig('artifacts/confusion_matrix.png')
```