Supplementary Material
Improving Animal Pose Estimation through Species Similarity Measures and Rigorous Label Definition

Abstract

This document contains supplementary material. Section 1 of this supplemental material contains the lists of identified similar species, for each of the target species we consider (e.g., antelope, chimpanzee, mouse, elephant, and black bear). Section 2 of the supplemental material contains code snippets to compute the automated species similarity measures.

1. Species Similarity Lists

This section contains the ranked similarity lists generated by the automated species similarity methods described in Section 3.1.

Method	**Top 10 Closest Species to Antelopes**
Taxonomic Similarity	Argali Sheep, Bison, Buffalo, Cow, Sheep, Deer, Moose
Centroid Variation	Argali Sheep, Horse, Deer, Zebra, Moose, Giraffe, King Cheetah, Sheep, Bison, Fox
Skeleton Ratios	Argali Sheep, Horse, Moose, Dog, Zebra, Deer, Sheep, Cow, Fox, Buffalo
ORB Descriptors	Rabbit, Deer, Zebra, Horse, Bison, Wolf, Buffalo, Rhino, Cat, Fox
DINOv2+ CLIP	Deer, Giraffe, Bison, Rabbit, Fox, Argali Sheep, Buffalo, Cheetah, Moose, Cow
Human Ranking	Deer, Moose, Zebra, Horse, Giraffe, Sheep, Cow, Argali Sheep, Bison, Buffalo

Table 1. Top 10 most similar species to antelopes selected by each method.

Method	**Top 10 Closest Species to Chimpanzees**
Centroid Variation	Uakari, Gorilla, Alouatta, Spider Monkey, Noisy Night Monkey, Monkey, Panda, Hamster, Polar Bear, Brown Bear
Skeleton Ratios	Monkey, Alouatta, Noisy Night Monkey, Marmot, Uakari, Bobcat, Spider Monkey, Brown Bear, Raccoon, Polar Bear
DINOv2+ CLIP	Spider Monkey, Elephant, Polar Bear, Rhino, Panda, Brown Bear, Gorilla, Otter, Bison, Noisy Night Monkey

Table 2. Top 10 most similar species to chimpanzees selected by each method.

Method	Top 10 Closest Species to Mice
Centroid Variation	Skunk, Rat, Wolf, Weasel, Rabbit, Snow Leopard, Beaver, Fox, Leopard, Jaguar
Skeleton Ratios	Bison, Horse, Zebra, Rat, Dog, Pig, Raccoon, Cow, Skunk, Sheep
DINOv2+ CLIP	Rat, Rabbit, Squirrel, Hamster, Beaver, Weasel, Skunk, Raccoon, Otter, Pig

Table 3. Top 10 most similar species to mice selected by each method.

Method	Top 10 Closest Species to Elephants
Centroid Variation	Rhino, Bison, Zebra, Sheep, Polar Bear, Cow, Moose, Cheetah, Horse, Brown Bear
Skeleton Ratios	Mouse, Horse, Rat, Zebra, Hamster, Pig, Rhino, Bison, Moose, Giraffe
DINOv2+ CLIP	Giraffe, Hippo, Panda, Buffalo, Spider Monkey, Zebra, Rhino, Cow, Lion, Leopard

Table 4. Top 10 most similar species to elephants selected by each method.

Method	Top 10 Closest Species to Black Bears
Centroid Variation	Buffalo, Brown Bear, Polar Bear, Weasel, Snow Leopard, Leopard, Cat, Jaguar, Wolf, Skunk
Skeleton Ratios	Sheep, Polar Bear, Moose, Snow Leopard, Brown Bear, Panda, Panther, Jaguar, Rhino, King Cheetah
DINOv2+ CLIP	Brown Bear, Polar Bear, Moose, Bison, Panther, Panda, Leopard, Jaguar, Tiger, Raccoon

Table 5. Top 10 most similar species to black bears selected by each method.

2. Species Similarity Metrics Code

This section provides the Python implementation for the species similarity metrics discussed in our paper.

2.1. DINOv2+CLIP Metric

The following code implements the similarity metric that combines features from DINOv2 and CLIP models and performs K-Nearest Neighbor comparison for the between the target species features and those of all other species in the dataset, to determine the most similar species, by those that appear most often in the K nearest neighbors.

2.1.1. Processing Dataset using DINOv2 and CLIP

This component calculates the feature DINOv2 and CLIP feature vectors for each image in the dataset.

```python
"""
This module implements a similarity metric based on features extracted using
DINOv2 and CLIP feature extraction models.
"""

# Process each image in the dataset with DINOv2 and CLIP
with torch.no_grad():
    for file in os.listdir(data_folder):
        if file.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.tiff')):
            file_path = os.path.join(data_folder, file)
            image_id = int(os.path.splitext(file)[0])

            # Get species and bbox info from annotations
            annotation = image_annotations.get(image_id)
            if not annotation:
                continue  # Skip if no annotation
            species_name = category_mapping[annotation["category_id"]]
            gt_bbox = annotation["bbox"]

            # Get keypoints
            original_keypoints = annotation.get("keypoints", [])
            num_keypoints = annotation.get("num_keypoints", 0)

            if num_keypoints == 0 or not original_keypoints: continue

            # Open image
            image = Image.open(file_path).convert("RGB")
            image_np = np.array(image)
            img_h, img_w = image_np.shape[:2]

            # Extract bounding box
            x_min, y_min, width, height = gt_bbox
            x_max, y_max = x_min + width, y_min + height
            bbox = [x_min, y_min, x_max, y_max]

            # --- 1. Cropping ---
            x_min, y_min, width, height = map(int, gt_bbox)
            x_min = max(0, x_min)
            y_min = max(0, y_min)
            x_max = min(img_w, x_min + width)
            y_max = min(img_h, y_min + height)
            crop_width = x_max - x_min
            crop_height = y_max - y_min

            if crop_width <= 0 or crop_height <= 0: continue

            cropped_image_np = image_np[y_min:y_max, x_min:x_max]
            cropped_image = Image.fromarray(cropped_image_np)
            original_cropped_width, original_cropped_height = cropped_image.size # Size AFTER crop

            # --- Store parameters needed for the mask function ---
            crop_origin = (x_min, y_min)
            original_crop_size = (original_cropped_width, original_cropped_height)
            # ----------------------------------------------------

            # --- 2. Resizing ---
            aspect_ratio = original_cropped_width / original_cropped_height
            if original_cropped_width >= original_cropped_height:
                new_width = img_size
                new_height = int(img_size / aspect_ratio)
            else:
                new_height = img_size
                new_width = int(img_size * aspect_ratio)
            new_width = max(1, new_width)
            new_height = max(1, new_height)

            resized_image = cropped_image.resize((new_width, new_height), Image.BICUBIC)
            resized_width, resized_height = resized_image.size

            # --- Store parameters needed for the mask function ---
            scale_x = resized_width / original_cropped_width if original_cropped_width > 0 else 1
            scale_y = resized_height / original_cropped_height if original_cropped_height > 0 else 1
            resize_scale = (scale_x, scale_y)
            # ----------------------------------------------------

            # --- 3. Padding ---
            left_pad = (img_size - resized_width) // 2
            top_pad = (img_size - resized_height) // 2
            right_pad = img_size - resized_width - left_pad
            bottom_pad = img_size - resized_height - top_pad

            padded_image = ImageOps.expand(resized_image, (left_pad, top_pad, right_pad, bottom_pad), fill='black')
            final_image_pil = padded_image
            final_image_np = np.array(final_image_pil) # Final 840x840 numpy array

            # --- Store parameters needed for the mask function ---
            padding_offset = (left_pad, top_pad)
            # ----------------------------------------------------

            # --- Create Masks around each Key Points ---
            # Pass the original keypoints and all transformation parameters
            keypoint_mask = create_keypoint_mask_transform_inside(
                final_image_shape=final_image_np.shape,
                original_keypoints=original_keypoints, # Pass original keypoints
                num_keypoints=num_keypoints,
                crop_origin=crop_origin,
                original_crop_size=original_crop_size,
                resize_scale=resize_scale,
                padding_offset=padding_offset
            )
            # ------------------------------------------

            # Convert image to grayscale
            grayscale_image = np.dot(final_image_np[...,:3], [0.2989, 0.5870, 0.1140])
            grayscale_image = grayscale_image / 255.0 * 0.9 + 0.1

            # Apply mask to grayscale image
            grayscale_image[~keypoint_mask] = 0
            grayscale_image = (grayscale_image * 255).astype(np.uint8)

            # Create masked image
            image_with_mask = final_image_np.copy()
            image_with_mask[~keypoint_mask] = 0
            image_with_mask[keypoint_mask] = np.expand_dims(grayscale_image, axis=-1)[keypoint_mask]
            final_image = Image.fromarray(image_with_mask)

            # Feature extraction
            img_t = transform1(final_image).to(device)
            features_dict = dinov2_vitl14.forward_features(img_t.unsqueeze(0))
            dinov2_features = features_dict['x_norm_patchtokens']

            image_input = clip_transform(image).unsqueeze(0).to(device)
            clip_features = clip_model.get_image_features(image_input)
            # Following Fusing DINO & SD paper for how to reshape CLIP features to match DINOv2
            clip_features_expanded = clip_features.unsqueeze(1).expand(1, dinov2_features.shape[1], -1)

            # Normalize DINOv2 and CLIP features - Following Fusing DINO & SD paper
            dinov2_features = dinov2_features / dinov2_features.norm(dim=-1, keepdim=True)
            clip_features_expanded = clip_features_expanded / clip_features_expanded.norm(dim=-1, keepdim=True)
            #clip_features_expanded = clip_features.unsqueeze(1).expand(-1, dinov2_features.shape[1], -1)

            combined_features = torch.cat([dinov2_features, clip_features_expanded], dim=-1)

            # Store features in species dictionary
            if species_name not in species_features:
```

```python
                species_features[species_name] = []
            species_features[species_name].append(combined_features)

# Convert lists to tensors
for species in species_features:
    species_features[species] = torch.cat(species_features[species], dim=0)
```

Listing 1. DINO+CLIP-based similarity metric implementation.

2.1.2. Species Similarity Analysis

This component performs the K-Nearest Neighbor for the nearest 10 neighbors.

```python
"""
This module implements a peforms the comparison among DINOv2+CLIP features extracted
from each species to determine the species most similar to the species of interest.
"""

comparison_species = "species"  # Species to compare against others

# Flatten and collect all species feature vectors (excluding antelope)
all_features = []
labels = []

for species_name, species_feature in species_features.items():
    if species_name == comparison_species:
        continue  # Skip comparison_species embeddings

    num_images = species_feature.shape[0]
    species_feature = species_feature.reshape(num_images, -1).cpu().numpy()
    all_features.append(species_feature)
    labels.extend([species_name] * num_images)

# Stack all non-comparison_species feature vectors into a single matrix
all_features = np.vstack(all_features)

# Get comparison_species features (to query k-NN)
comparison_species_features = species_features[comparison_species].reshape(-1, all_features.shape
    [1]).cpu().numpy()

# Train k-NN only on non-comparison_species species
knn = NearestNeighbors(n_neighbors=10, metric="cosine")  # Adjust k as needed
knn.fit(all_features)

# Find the k nearest neighbors for comparison_species images (excluding other comparison_speciess)
distances, indices = knn.kneighbors(comparison_species_features)

# Count occurrences of nearest species, this gives you the ranked order of most similar species
nearest_species = Counter([labels[i] for i in indices.flatten()])
```

Listing 2. DINOv2+CLIP-based similarity analysis.

2.2. Centroid Variation Metric

The following code implements the centroid variation methodology which analyzes species similarity based on keypoint distance variations from centroids.

```python
"""
Centroid Variation Metric for Species Similarity Analysis.

This module computes keypoint variations from centroids and analyzes species similarity
based on these variations using cosine similarity.
"""
import numpy as np
import pandas as pd
from sklearn.metrics.pairwise import cosine_similarity

def compute_keypoint_variations(keypoints):
    """
    Calculate coefficient of variation (CV) for keypoint distances from the centroid.

    Parameters:
    -----------
    keypoints : list
        Flattened keypoints array with [x, y, visibility] for each keypoint

    Returns:
    --------
    numpy.ndarray
        Array of coefficient of variation for each keypoint,
        or -np.inf for invisible keypoints
    """
    keypoints = np.array(keypoints).reshape(-1, 3)
    visible = keypoints[:, 2] > 0  # Mask for visible keypoints

    if not np.any(visible):
        return np.full(len(keypoints), -np.inf)  # No visible keypoints

    # Calculate centroid using only visible keypoints
    centroid = np.mean(keypoints[visible, :2], axis=0)

    # Calculate distances from centroid
    distances = np.full(len(keypoints), -np.inf)  # Default to -inf for invisible keypoints
    distances[visible] = np.linalg.norm(keypoints[visible, :2] - centroid, axis=1)

    # If all distances are zero, return zeros to avoid division by zero
    if np.all(distances[visible] == 0):
        variations = np.zeros(len(keypoints))
        variations[~visible] = -np.inf
        return variations

    # Calculate coefficient of variation (CV)
    # CV = normalized distance from mean (variation of standard CV)
    visible_distances = distances[visible]
    if np.mean(visible_distances) == 0:
        variations = np.zeros(len(keypoints))
    else:
        variations = np.zeros(len(keypoints))
        variations[visible] = visible_distances / np.mean(visible_distances)

    # Set non-visible keypoints to -inf
    variations[~visible] = -np.inf

    return variations

def compute_species_similarity(species_variations):
    """
    Compute cosine similarity between species based on keypoint variations.

    Parameters:
    -----------
    species_variations : dict
        Dictionary with species names and their average keypoint variations

    Returns:
    --------
    pd.DataFrame
        Similarity matrix with species names as index and columns
    """
    # Convert variations to a 2D numpy array
    species_names = list(species_variations.keys())
    variations_array = np.array(list(species_variations.values()))

    # Compute cosine similarity
    similarity_matrix = cosine_similarity(variations_array)

    # Create a DataFrame with species names
    similarity_df = pd.DataFrame(
        similarity_matrix,
        index=species_names,
        columns=species_names
    )

    return similarity_df

def find_closest_to_antelope(similarity_df, top_n=100):
    """
    Find species most similar to antelope based on cosine similarity.

    Parameters:
    -----------
    similarity_df : pd.DataFrame
        Cosine similarity matrix
    top_n : int, optional
        Number of top similar species to return

    Returns:
    --------
    tuple
        Top N and bottom N species most similar to antelope, sorted by similarity
    """
    antelope_similarities = similarity_df.loc['antelope'].sort_values(ascending=False)

    # Exclude self-similarity (it is 1.0)
    antelope_similarities = antelope_similarities[
        (antelope_similarities < 1.0) & (antelope_similarities != 0.9999999999999999)
    ]

    return antelope_similarities.head(top_n), antelope_similarities.tail(top_n)
```

Listing 3. Centroid variation metric implementation.

2.3. ORB Metric

The following code snippet implements the similarity metric based on Oriented FAST and Rotated BRIEF (ORB) features.

```python
"""
ORB-based similarity metric implementation.

This module implements a similarity metric based on ORB (Oriented FAST and Rotated BRIEF)
features extracted using OpenCV.
"""


class ORBSimilarity(SimilarityMetric):
    """Similarity metric based on ORB features extracted from images."""

    def __init__(self):
        """Initialize the ORB similarity metric."""
        super().__init__(name="orb_similarity")
        self.species_features = {}  # Feature distributions per species
        self.similarity_matrix = None

    def extract_features_from_patch(self, gray_img: np.ndarray, keypoint: Tuple[float, float],
                                    orb: cv2.ORB, patch_sizes: List[int] = [48, 96, 144]) -> np.ndarray:
        """Extract multi-scale ORB features from patches around a keypoint.

        Args:
            gray_img: Grayscale image
            keypoint: (x, y) coordinates of the keypoint
            orb: ORB detector instance
            patch_sizes: List of patch sizes to use for multi-scale analysis

        Returns:
            Combined feature vector or None if extraction fails
        """
        x, y = keypoint
        all_descriptors = []

        # Get image dimensions for relative position encoding
        img_height, img_width = gray_img.shape
        # Encode relative position of keypoint
        rel_x, rel_y = x / img_width, y / img_height

        for patch_size in patch_sizes:
            half_size = patch_size // 2
            x1, y1 = max(0, int(x) - half_size), max(0, int(y) - half_size)
            x2, y2 = min(gray_img.shape[1], int(x) + half_size), min(gray_img.shape[0], int(y) + half_size)
            patch = gray_img[y1:y2, x1:x2]

            if patch.size == 0 or patch.shape[0] < 32 or patch.shape[1] < 32:
                continue

            # Resize patch to fixed size for consistent feature extraction
            patch = cv2.resize(patch, (64, 64))

            # Enhance contrast to better capture structure
            patch = cv2.equalizeHist(patch)

            # Detect and compute features
            kps = orb.detect(patch, None)
            if kps:
                # Sort by response strength and take top N
                kps = sorted(kps, key=lambda kp: kp.response, reverse=True)[:3]
                _, descs = orb.compute(patch, kps)

                if descs is not None and len(descs) > 0:
                    # Convert binary descriptors to float for averaging
                    descs = descs.astype(np.float32)
                    # Add relative position information to make features location-aware
                    pos_feature = np.array([rel_x, rel_y] * (descs.shape[1] // 2))
                    descs = descs * (1 - pos_feature) + pos_feature
                    all_descriptors.append(descs)

        if not all_descriptors:
            return None

        # Stack all descriptors and compute mean
        all_descs = np.vstack(all_descriptors)
        return np.mean(all_descs, axis=0)

    def process_annotations(self, annotation_files: List[str],
                           base_path: str, output_dir: str, use_existing_features: bool = False) -> Dict:
        """Process annotation files to extract ORB features for each species.

        Args:
            annotation_files: List of annotation JSON filenames
            base_path: Base directory path for annotations
            output_dir: Directory to save output feature CSV files
            use_existing_features: If True, try to load existing features first

        Returns:
            Dictionary of species data with ORB features
        """
        # Try to load existing features if requested
        if use_existing_features:
            try:
                print("Attempting to load existing features...")
                loaded_features = self.load_species_features(output_dir)
                if loaded_features:
                    print("Successfully loaded existing features.")
                    self.species_features = loaded_features
                    return self.species_features
            except Exception as e:
                print(f"Failed to load existing features: {e}")
                print("Falling back to computing new features...")

        os.makedirs(output_dir, exist_ok=True)

        # Create ORB detector with better parameters for anatomical features
        orb = cv2.ORB_create(
            nfeatures=3000,    # More features for better coverage
            scaleFactor=1.1,   # Finer scale progression
            nlevels=12,        # More scale levels
            edgeThreshold=15,  # More sensitive edge detection
            patchSize=31,      # Larger patch size
            fastThreshold=20,  # More sensitive to corners
            firstLevel=0,
            WTA_K=3           # More discriminative features
        )

        # Dictionary to store features for each species
        species_features_dict = defaultdict(list)
        exclude_images = set()  # Images with multiple annotations

        # First pass to identify multi-annotation images
        print("Scanning for multi-annotation images...")
        for filename in annotation_files:
            filepath = os.path.join(base_path, filename)
            if not os.path.exists(filepath):
                print(f"Warning: Annotation file not found: {filepath}")
                continue

            try:
                with open(filepath, 'r') as file:
                    data = json.load(file)

                counts = defaultdict(int)
                if 'annotations' in data:
                    for ann in data['annotations']:
                        counts[ann['image_id']] += 1
                exclude_images.update({img_id for img_id, c in counts.items() if c > 1})
            except Exception as e:
                print(f"Error reading {filepath}: {e}")

        print(f"Found {len(exclude_images)} images with multiple annotations to exclude.")

        # Process annotations and extract features
        print("Extracting multi-scale ORB features...")
        processed_files = 0
        processed_images = 0

        for filename in annotation_files:
            filepath = os.path.join(base_path, filename)
            if not os.path.exists(filepath):
                continue

            try:
                with open(filepath, 'r') as file:
                    data = json.load(file)
                processed_files += 1

                category_map = {c['id']: c['name'] for c in data['categories']}
                image_map = {img['id']: img['file_name'] for img in data['images']}

                for ann in data['annotations']:
                    if ann['image_id'] in exclude_images or ann.get('iscrowd', 0) == 1:
                        continue

                    species = category_map.get(ann['category_id'])
                    if not species:
                        continue

                    image_filename = image_map.get(ann['image_id'])
                    if not image_filename:
                        continue

                    img_path = os.path.join(base_path, '..', 'data', image_filename)
                    if not os.path.exists(img_path):
                        continue

                    img = cv2.imread(img_path)
                    if img is None:
                        continue

                    # Convert to grayscale with better contrast
                    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
                    gray = cv2.equalizeHist(gray)

                    keypoints = ann.get('keypoints', [])
                    if not keypoints or len(keypoints) < 3:
                        continue

                    # Extract features for each valid keypoint
                    keypoint_features = []
                    for kp_idx in range(0, len(keypoints), 3):
                        x, y, v = keypoints[kp_idx:kp_idx+3]

                        if v == 0 or x < 0 or y < 0 or x >= gray.shape[1] or y >= gray.shape[0]:
                            continue

                        features = self.extract_features_from_patch(gray, (x, y), orb)
                        if features is not None:
                            keypoint_features.append(features)

                    if keypoint_features:
                        # Weight features by their anatomical importance
                        weights = np.ones(len(keypoint_features))
                        # Give more weight to core body keypoints (if available)
                        core_indices = [0, 1, 2, 3]  # Adjust these indices based on your keypoint schema
                        for idx in core_indices:
                            if idx < len(weights):
                                weights[idx] = 2.0
                        weights = weights / np.sum(weights)

                        # Compute weighted average of features
                        image_features = np.average(keypoint_features, axis=0, weights=weights)
                        species_features_dict[species].append(image_features)

                    processed_images += 1
                    if processed_images % 100 == 0:
                        print(f"Processed {processed_images} images...")

            except Exception as e:
                print(f"Error processing {filepath}: {e}")
                import traceback
                traceback.print_exc()

        print(f"\nProcessed {processed_images} images across {processed_files} files.")

        # Combine all features for PCA
        print("\nApplying PCA to all features...")
        all_features = []
        species_indices = []  # Keep track of which features belong to which species

        for species, features in species_features_dict.items():
            if not features:
                continue
            features_array = np.array(features)
            all_features.append(features_array)
            species_indices.extend([species] * len(features_array))

        if not all_features:
            raise ValueError("No features extracted from any species")

        # Stack all features and apply PCA globally
        all_features_array = np.vstack(all_features)
        from sklearn.decomposition import PCA
        pca = PCA(n_components=32)  # Fixed number of components for all species
        all_features_transformed = pca.fit_transform(all_features_array)

        # Split features back by species
        current_idx = 0
        for species in species_features_dict.keys():
            if not species_features_dict[species]:
                continue

            n_samples = len(species_features_dict[species])
            species_features = all_features_transformed[current_idx:current_idx + n_samples]
            current_idx += n_samples

            # Store feature distribution statistics
            self.species_features[species] = {
                'mean': np.mean(species_features, axis=0),
                'std': np.std(species_features, axis=0),
                'distribution': species_features
            }

            # Save features to file
            output_file = os.path.join(output_dir, f"{species}_orb_features.csv")
            pd.DataFrame(species_features).to_csv(output_file, index=False)
            print(f"Saved features for {species} to {output_file}")

        return self.species_features

    def compute_similarity(self) -> pd.DataFrame:
        """Compute similarity between species using Earth Mover's Distance (EMD)
        and distribution statistics.

        Returns:
            DataFrame containing pairwise similarities between species
        """
        if not self.species_features:
            raise ValueError("No features loaded. Run process_annotations or load_species_features
    first.")

        species_list = list(self.species_features.keys())
        n_species = len(species_list)
        similarity_matrix = np.zeros((n_species, n_species))

        for i, species1 in enumerate(species_list):
            for j, species2 in enumerate(species_list):
                if i == j:
                    similarity_matrix[i, j] = 1.0
                    continue

                # Get feature distributions
                dist1 = self.species_features[species1]
                dist2 = self.species_features[species2]

                # Compute EMD between feature distributions
                # Take mean across feature dimensions to get a 1D distribution per species
                dist1_mean = np.mean(dist1['distribution'], axis=1)
                dist2_mean = np.mean(dist2['distribution'], axis=1)

                # Normalize distributions for EMD
                dist1_mean = dist1_mean / np.sum(dist1_mean) if np.sum(dist1_mean) != 0 else
    dist1_mean
                dist2_mean = dist2_mean / np.sum(dist2_mean) if np.sum(dist2_mean) != 0 else
    dist2_mean

                # Compute EMD
                emd_dist = wasserstein_distance(dist1_mean, dist2_mean)

                # Compute mean feature similarity with std weighting
                mean_sim = cosine_similarity(
                    dist1['mean'].reshape(1, -1),
                    dist2['mean'].reshape(1, -1)
                )[0, 0]

                # Weight similarity by standard deviation overlap
                std_overlap = np.mean(np.minimum(dist1['std'], dist2['std']) / np.maximum(dist1['std
    '], dist2['std']))

                # Combine metrics with stronger EMD weight and std penalty
                emd_sim = 1 / (1 + 5 * emd_dist)  # Increase EMD impact
                similarity = 0.6 * emd_sim + 0.3 * mean_sim + 0.1 * std_overlap

                # Apply sigmoid to spread out similarity scores
                similarity = 1 / (1 + np.exp(-10 * (similarity - 0.5)))
                similarity_matrix[i, j] = similarity

        self.similarity_matrix = pd.DataFrame(
            similarity_matrix,
            index=species_list,
            columns=species_list
        )

        return self.similarity_matrix
```

Listing 4. ORB-based similarity metric implementation.

2.4. Skeleton Ratios Metric

The following code implements the skeleton ratio analysis method for animal pose estimation, which computes similarity based on normalized skeleton ratios derived from keypoint connections.

```python
"""
Limb Ratio Analysis Method for Animal Pose Estimation
Core computational method for supplement documentation
"""

from math import sqrt

# AP10K skeleton definition - 17 keypoint connections
SKELETON = [
    [1, 2], [1, 3], [2, 3], [3, 4], [4, 5], [4, 6], [6, 7], [7, 8],
    [4, 9], [9, 10], [10, 11], [5, 12], [12, 13], [13, 14],
    [5, 15], [15, 16], [16, 17]
]

def calculate_limb_ratios(keypoints, bbox):
    """
    Calculate normalized limb ratios from keypoint annotations.

    Args:
        keypoints: List of [x, y, visibility] for each keypoint (51 values total)
        bbox: Bounding box [x, y, width, height]

    Returns:
        dict: Limb ratios normalized by bounding box height
    """
    bbox_height = bbox[3]
    if bbox_height <= 0:
        return {}

    limb_ratios = {}

    for connection in SKELETON:
        kp1_idx = connection[0] - 1  # Convert to 0-based indexing
        kp2_idx = connection[1] - 1

        # Extract keypoint coordinates and visibility
        x1, y1, v1 = keypoints[kp1_idx*3], keypoints[kp1_idx*3+1], keypoints[kp1_idx*3+2]
        x2, y2, v2 = keypoints[kp2_idx*3], keypoints[kp2_idx*3+1], keypoints[kp2_idx*3+2]

        # Only calculate if both keypoints are visible
        if v1 > 0 and v2 > 0:
            # Euclidean distance between keypoints
            distance = sqrt((x2 - x1)**2 + (y2 - y1)**2)
            # Normalize by bounding box height
            ratio = distance / bbox_height
            limb_ratios[f"limb_{connection[0]}_{connection[1]}"] = round(ratio, 4)

    return limb_ratios

def compute_species_average_ratios(species_data):
    """
    Compute average limb ratios for a species from multiple samples.

    Args:
        species_data: List of limb ratio dictionaries for the species

    Returns:
        dict: Average limb ratios for the species
    """
    if not species_data:
        return {}

    # Get all possible limb connections
    all_limbs = set()
    for sample in species_data:
        all_limbs.update(sample.keys())

    species_averages = {}
    for limb in all_limbs:
        values = [sample[limb] for sample in species_data if limb in sample]
        if values:
            species_averages[limb] = sum(values) / len(values)

    return species_averages

def cosine_similarity(vector_a, vector_b):
    """
    Calculate cosine similarity between two limb ratio vectors.

    Args:
        vector_a, vector_b: Lists of limb ratio values

    Returns:
        float: Cosine similarity score (0-1)
    """
    dot_product = sum(a * b for a, b in zip(vector_a, vector_b))
    magnitude_a = sqrt(sum(a ** 2 for a in vector_a))
    magnitude_b = sqrt(sum(b ** 2 for b in vector_b))

    if magnitude_a == 0 or magnitude_b == 0:
        return 0.0

    return dot_product / (magnitude_a * magnitude_b)

def build_similarity_matrix(species_averages):
    """
    Build pairwise cosine similarity matrix between species.

    Args:
        species_averages: Dict mapping species names to average limb ratios

    Returns:
        tuple: (species_names, similarity_matrix)
    """
    species_names = list(species_averages.keys())

    # Get all limb types across all species
    all_limbs = sorted(set(limb for avg in species_averages.values() for limb in avg.keys()))

    # Convert to vectors with consistent ordering
    species_vectors = {}
    for species, averages in species_averages.items():
        vector = [averages.get(limb, 0.0) for limb in all_limbs]
        species_vectors[species] = vector

    # Compute pairwise similarities
    similarity_matrix = []
    for species_a in species_names:
        row = []
        vector_a = species_vectors[species_a]
        for species_b in species_names:
            vector_b = species_vectors[species_b]
            similarity = cosine_similarity(vector_a, vector_b)
            row.append(round(similarity, 4))
        similarity_matrix.append(row)

    return species_names, similarity_matrix
```

Listing 5. Skeleton ratio analysis method implementation.