Week 7 Report - NFHM

Romouald Dombrovski

October 03, 2024

1. Time Log

What progress did you make last week?

- Added Euclidean distance as a metric to model eval
- Added Arborclip as another model to compare
- Vectorized the comparison calculations for larger metrics evaluation (30,000 images)
- Produced visual representation for model evaluation comparison

2. Abstract

Nguyen, D.M.H., et al. 2023. LVM-Med: Learning Large-Scale Self-Supervised Vision Models for Medical Imaging via Second-order Graph Matching. *arXiv:2306.11925v3*. http://doi.org/10.48550/arXiv.2306.11925

Abstract: Obtaining large pre-trained models that can be fine-tuned to new tasks with limited annotated samples has remained an open challenge for medical imaging data. While pre-trained deep networks on ImageNet and vision-language foundation models trained on web-scale data are prevailing approaches, their effectiveness on medical tasks is limited due to the significant domain shift between natural and medical images. To bridge this gap, we introduce LVM-Med, the first family of deep networks trained on large-scale medical datasets. We have collected approximately 1.3 million medical images from 55 publicly available datasets, covering a large number of organs and modalities such as CT, MRI, X-ray, and Ultrasound. We benchmark several state-of-the-art self-supervised algorithms on this dataset and propose a novel self-supervised contrastive learning algorithm using a graph-matching formulation. The proposed approach makes three contributions: (i) it integrates prior pair-wise image similarity metrics based on local and global information; (ii) it captures the structural constraints of feature embeddings through a loss function constructed via a combinatorial graph-matching objective; and (iii) it can be trained efficiently end-to-end using modern gradient-estimation techniques for black-box solvers. We thoroughly evaluate the proposed LVM-Med on 15 downstream medical tasks ranging from segmentation and classification to object detection, and both for the in and out-of-distribution

settings. LVM-Med empirically outperforms a number of state-of-the-art supervised, self-supervised, and foundation models. For challenging tasks such as Brain Tumor Classification or Diabetic Retinopathy Grading, LVM-Med improves previous vision-language models trained on 1 billion masks by 6-7% while using only a ResNet-50.

Summary (GPT-4o)**:** The paper titled *"LVM-Med: Learning Large-Scale Self-Supervised Vision Models for Medical Imaging via Second-order Graph Matching"* introduces LVM-Med, a novel self-supervised learning (SSL) model specifically designed for medical imaging tasks. The authors address the limitations of pre-trained models on natural images when applied to the medical domain due to domain shifts. LVM-Med is trained on a curated dataset of 1.3 million medical images from 55 publicly available datasets covering various organs and imaging modalities such as CT, MRI, X-ray, and Ultrasound. The key innovation of LVM-Med is its use of second-order graph matching, which integrates both local and global image information to enhance feature embeddings. The model is evaluated on 15 downstream medical tasks, including segmentation and classification, demonstrating superior performance over state-of-the-art supervised and self-supervised models like ResNet-50 and Vision Transformer (ViT). Furthermore, LVM-Med outperforms foundation models such as CLIP and SAM by up to 7% on challenging tasks like brain tumor classification and diabetic retinopathy grading. The proposed approach highlights the potential of LVM-Med in improving medical image analysis and presents a significant step toward creating large-scale, self-supervised vision models tailored for medical data.

3. Scripts and Code Blocks

Updated the stratification split

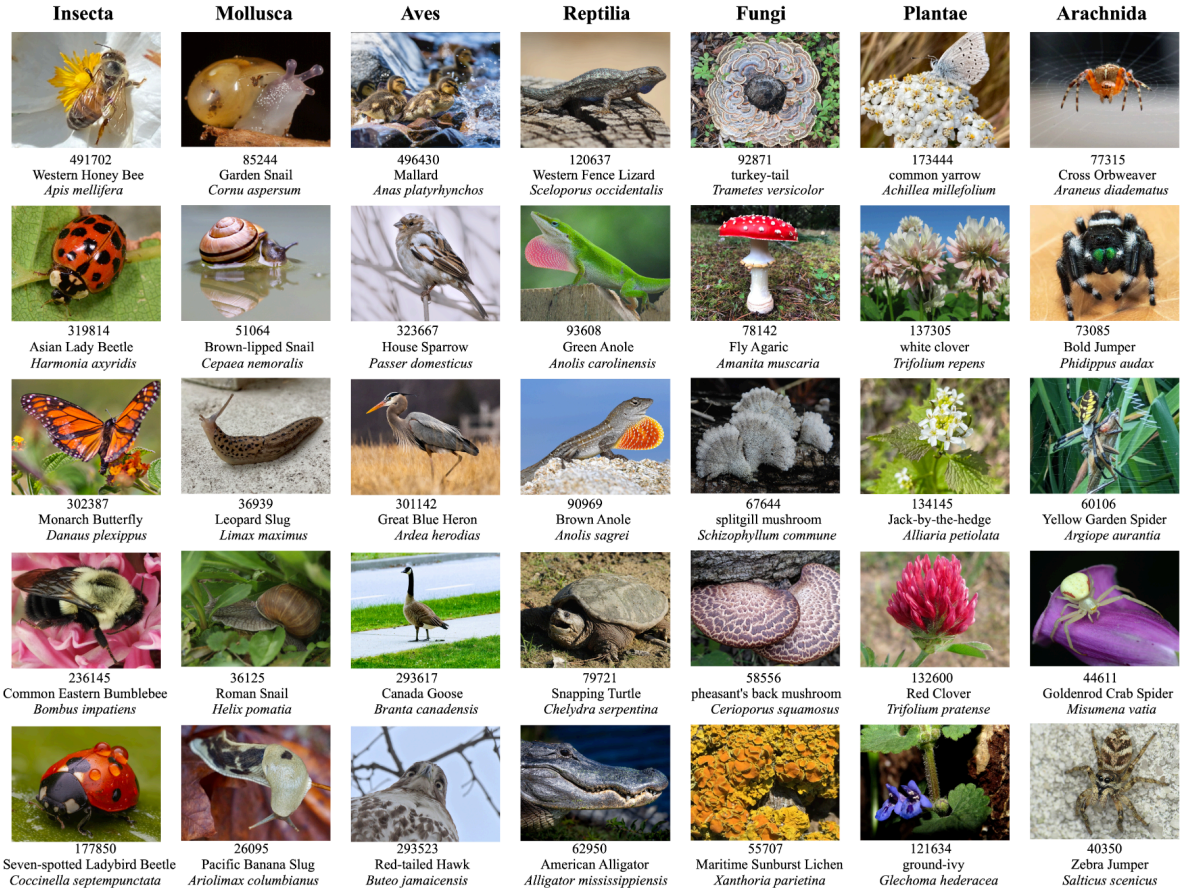
```
[ ]  from sklearn.model_selection import train_test_split

    def get_test_train_split(df_cleaned, test_size=0.1, stratify_by_scientific_name=False):
      if stratify_by_scientific_name:
        # if we want to get at least one from each we can filter out all options with only 1 image
        species_counts = df_cleaned['scientificName'].value_counts()
        valid_species = species_counts[species_counts > 1].index
        df_filtered = df_cleaned[df_cleaned['scientificName'].isin(valid_species)]
        # with stratification, we need to specify the test_size because we need to hit a minimum
        calc_min = len(valid_species)/len(df_filtered)
        min_split = max(test_size, calc_min)
        print(f'The split for stratification is: {min_split}')

        train_df, test_df = train_test_split(df_filtered, test_size=min_split, stratify=df_filtered['scientificName'])
      else:
        train_df, test_df = train_test_split(df_cleaned, test_size=test_size)
      return train_df, test_df
```

Added Arborclip (loading using weights, from bioclip endpoint)

```
elif model_name == 'arborclip':
  model_name = "ChihHsuan-Yang/ArborCLIP"
  filename = "arborclip-vit-b-16-from-bioclip-epoch-8.pt"

  # Download the file from the repository
  weights_path = hf_hub_download(repo_id=model_name, filename=filename)

  # Initialize the base BioCLIP model using OpenCLIP
  model, _, processing = open_clip.create_model_and_transforms('hf-hub:imageomics/bioclip')

  # Load the fine-tuned weights into the model
  checkpoint = torch.load(weights_path, map_location='cpu')

  if "state_dict" in checkpoint:
      state_dict = checkpoint["state_dict"]
  else:
      state_dict = checkpoint

  state_dict = {k.replace("module.", ""): v for k, v in state_dict.items()}
  missing_keys, unexpected_keys = model.load_state_dict(state_dict, strict=False)
```

The above changes were mainly to fix the way stratification was meant to work, the maximum was taken for the split size instead of the minimum (especially with larger datasets, the split may become 0.05. We want a decent stratified sample). Arborclip was another interesting model, built off the clip backbone and trained on further

animalia species photos:



Now with 4 models (Florence-2, Openclip, Bioclip, Arborclip), we wanted to run the evaluation metrics using all 30,000 images from the VLM4Bio dataset, to compare the contextual representation of the image embeddings created from these models.

Looking at the pgvector documentation (https://github.com/pgvector/pgvector), we saw that Euclidean distance was another popular metric to compare embedding vectors. So we added that on top of cosine similarity as a metric:

```
def calculate_similarity_and_distance(test_embeddings, train_embeddings, model):

    # we don't need the dim=1, in both cases its 1
    test_embeddings, train_embeddings = np.squeeze(test_embeddings, axis=1), np.squeeze(train_embeddings, axis=1)

    # if florence, avg across new dim 1, size = 577 (patches on img)
    if model == 'florence':
        test_embeddings = np.mean(test_embeddings, axis=1)
        train_embeddings = np.mean(train_embeddings, axis=1)

    similarity_matrix = cosine_similarity(test_embeddings, train_embeddings)
    distance_matrix = euclidean_distances(test_embeddings, train_embeddings)

    return similarity_matrix, distance_matrix
```

As the datasets were large (~30,000 embeddings), the comparison was vectorized

```python
def find_best_match_vectorized(test_embeddings, train_df, label_column, model):
    # Get train embeddings and labels as numpy arrays
    train_embeddings = np.array(train_df['image_embeddings'].tolist())
    train_labels = train_df[label_column].values

    # Calculate similarity and distance for all test embeddings at once
    similarity_matrix, distance_matrix = calculate_similarity_and_distance(test_embeddings, train_embeddings, model)

    # Get the labels with the highest individual similarity score and lowest distance
    highest_similarity_idx = similarity_matrix.argmax(axis=1)
    highest_similarity_labels = train_labels[highest_similarity_idx]
    highest_similarity_scores = similarity_matrix.max(axis=1)

    lowest_dist_idx = distance_matrix.argmin(axis=1)
    lowest_dist_labels = train_labels[highest_similarity_idx]
    lowest_dist_scores = similarity_matrix.min(axis=1)

    # create a repeated matrix of train labels to match the size of similarity matrix (test_size x train_size)
    repeated_train_labels = np.tile(train_labels, (similarity_matrix.shape[0], 1))

    # Create a DataFrame with each test embedding's comparison to all train labels
    # Flatten the similarity matrix and repeated labels to match
    similarity_df = pd.DataFrame({
        'test_idx': np.repeat(range(similarity_matrix.shape[0]), similarity_matrix.shape[1]),
        label_column: repeated_train_labels.flatten(),
        'similarity_score': similarity_matrix.flatten()
    })
    avg_similarity_df = similarity_df.groupby(['test_idx', label_column], as_index=False).mean()
    idx_max_avg = avg_similarity_df.groupby('test_idx')['similarity_score'].idxmax()
    highest_avg_sim_labels = avg_similarity_df.loc[idx_max_avg, label_column].values
    highest_avg_sim_scores = avg_similarity_df.loc[idx_max_avg, 'similarity_score'].values

    distance_df = pd.DataFrame({
        'test_idx': np.repeat(range(similarity_matrix.shape[0]), similarity_matrix.shape[1]),
        label_column: repeated_train_labels.flatten(),
        'distance_score': distance_matrix.flatten()
    })
    avg_dist_df = distance_df.groupby(['test_idx', label_column], as_index=False).mean()
    idx_min_avg = avg_dist_df.groupby('test_idx')['distance_score'].idxmin()
    lowest_avg_dist_labels = avg_dist_df.loc[idx_min_avg, label_column].values
    lowest_avg_dist_scores = avg_dist_df.loc[idx_min_avg, 'distance_score'].values

    # return highest_similarity_labels, highest_similarity_scores, highest_summed_label, highest_summed_score
    return highest_similarity_labels,highest_similarity_scores,highest_avg_sim_labels,highest_avg_sim_scores,lowest_dist_la
```

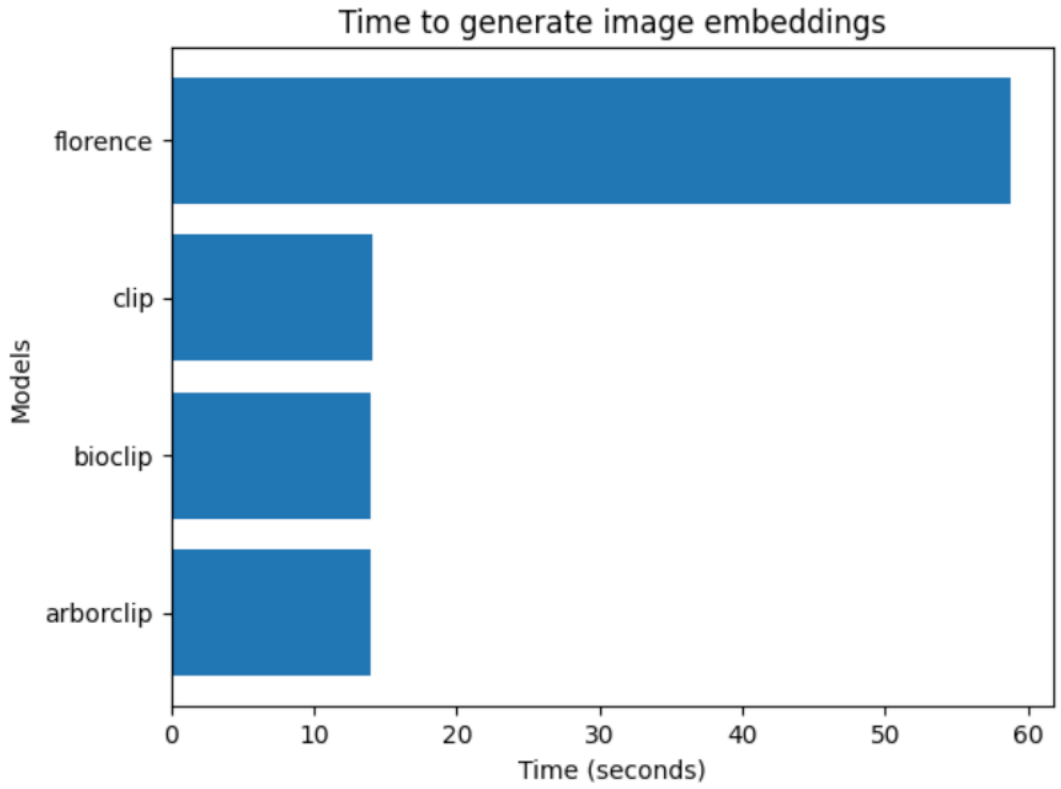
```python
def apply_best_match_vectorized(test_df, train_df, label_column, model):
    # Convert all test image embeddings to numpy arrays
    test_embeddings = np.array(test_df['image_embeddings'].tolist())

    # Vectorized function to find the best match for each test embedding
    highest_individual_name, highest_individual_score, highest_mean_name, highest_mean_score,lowest_dist_name, lowest_dist
        test_embeddings, train_df, label_column, model
    )

    # Assign results back to the test DataFrame
    test_df['highest_individual_name'] = highest_individual_name
    test_df['highest_individual_score'] = highest_individual_score
    test_df['highest_mean_name'] = highest_mean_name
    test_df['highest_mean_score'] = highest_mean_score
    test_df['lowest_dist_name'] = lowest_dist_name
    test_df['lowest_dist_score'] = lowest_dist_score
    test_df['lowest_avg_dist_name'] = lowest_avg_dist_name
    test_df['lowest_avg_dist_score'] = lowest_avg_dist_score

    # Calculate accuracy metrics
    accuracy_individual = np.mean(test_df['highest_individual_name'] == test_df[label_column])
    accuracy_mean = np.mean(test_df['highest_mean_name'] == test_df[label_column])
    accuracy_dist = np.mean(test_df['lowest_dist_name'] == test_df[label_column])
    accuracy_avg_dist = np.mean(test_df['lowest_avg_dist_name'] == test_df[label_column])

    print(f"Accuracy for model {model} on column {label_column} based on highest individual cosine similarity: {accuracy_
    print(f"Accuracy for model {model} on column {label_column} based on highest mean cosine similarity: {accuracy_mean *
    print(f"Accuracy for model {model} on column {label_column} based on lowest individual euclidean distance: {accuracy_
    print(f"Accuracy for model {model} on column {label_column} based on lowest mean euclidean distance: {accuracy_mean *

    return test_df, accuracy_individual, accuracy_mean, accuracy_dist, accuracy_avg_dist
```

After obtaining all the relevant information in a df, created graphs to demonstrate the results:

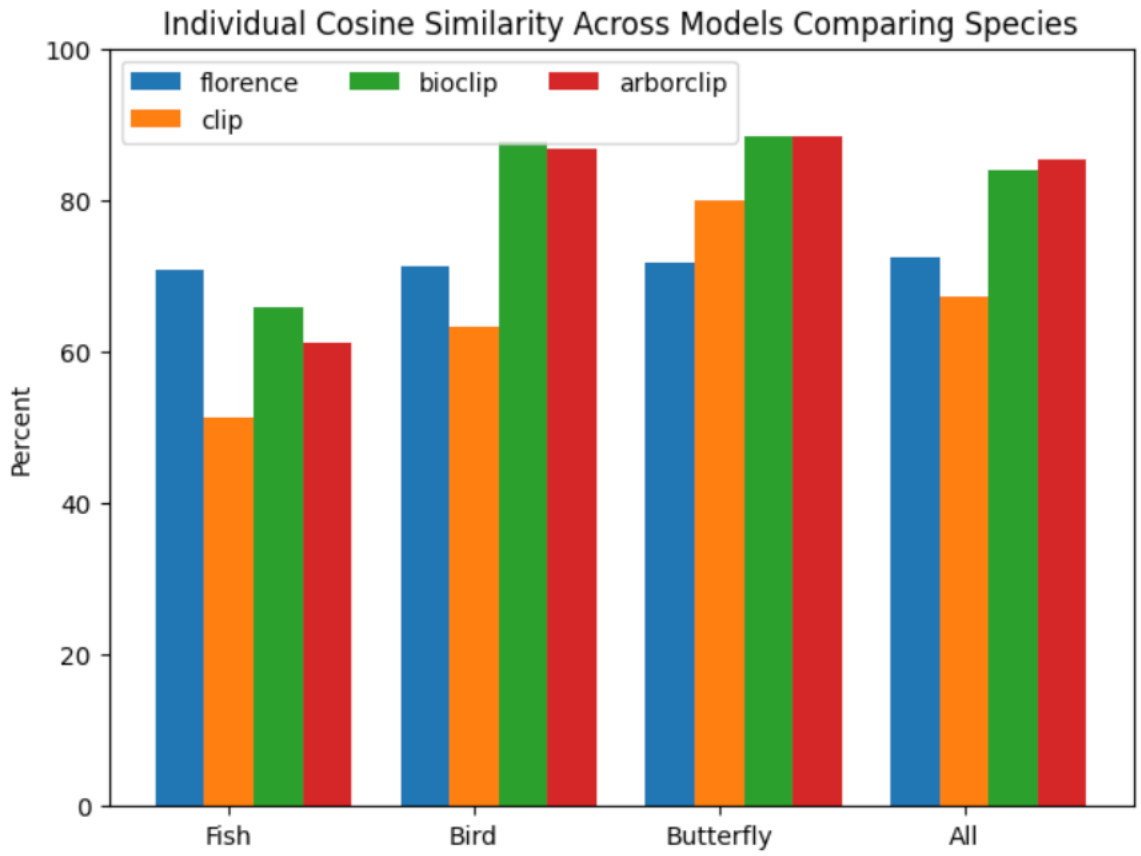
```
def make_plot(comparison_metric, df, column = 'species'):
  if comparison_metric == 'cosine':
    col1, col2 = 'accuracy_individual', 'accuracy_avg'
    title1, title2 = 'Individual Cosine Similarity', 'Average Cosine Similarity'
  elif comparison_metric == 'distance':
    col1, col2 = 'distance_individual', 'distance_avg'
    title1, title2 = 'Individual Euclidean Distance', 'Average Euclidean Distance'
  else:
    print('comparison metric is either distance or cosine')

  taxa = list(df['taxa'].unique())
  models = list(df['model'].unique())

  model_ind = {
      m: list(df.loc[df['model'] == m, col1].str.replace('%', '').astype(float)) for m in models
  }
  model_mean = {
      m: list(df.loc[df['model'] == m, col2].str.replace('%', '').astype(float)) for m in models
  }

  x = np.arange(len(taxa))  # the label locations
  width = 0.2  # the width of the bars
  multiplier = 0

  fig, ax = plt.subplots(layout='constrained')

  for model, measurement in model_ind.items():
      offset = width * multiplier
      rects = ax.bar(x + offset, measurement, width, label=model)
      multiplier += 1

  ax.set_ylabel('Percent')
  ax.set_title(f"{title1} Across Models Comparing {column.title()}")
  ax.set_xticks(x + width, taxa)
  ax.legend(loc='upper left', ncols=3)
  ax.set_ylim(0, 100)

  plt.show()

  fig, ax = plt.subplots(layout='constrained')
  multiplier = 0
  for model, measurement in model_mean.items():
      offset = width * multiplier
      rects = ax.bar(x + offset, measurement, width, label=model)
      multiplier += 1
  ax.set_ylabel('Percent')
  ax.set_title(f"{title2} Across Models Comparing {column.title()}")
  ax.set_xticks(x + width, taxa)
  ax.legend(loc='upper left', ncols=3)
  ax.set_ylim(0, 100)
  plt.show()

make_plot('cosine', species_rows, column='species')
make_plot('distance', species_rows, column='species')
make_plot('cosine', genus_rows, column='genus')
make_plot('distance', genus_rows, column='genus')
```

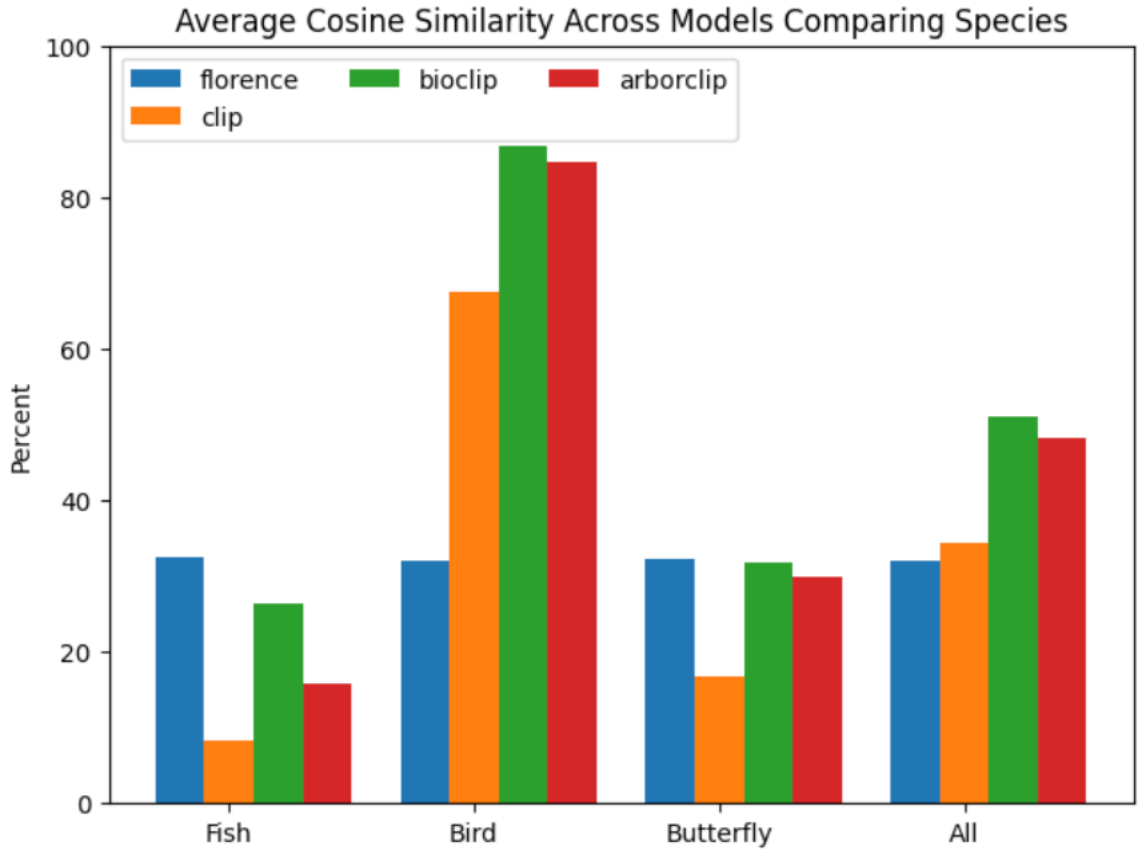
4. Visualization

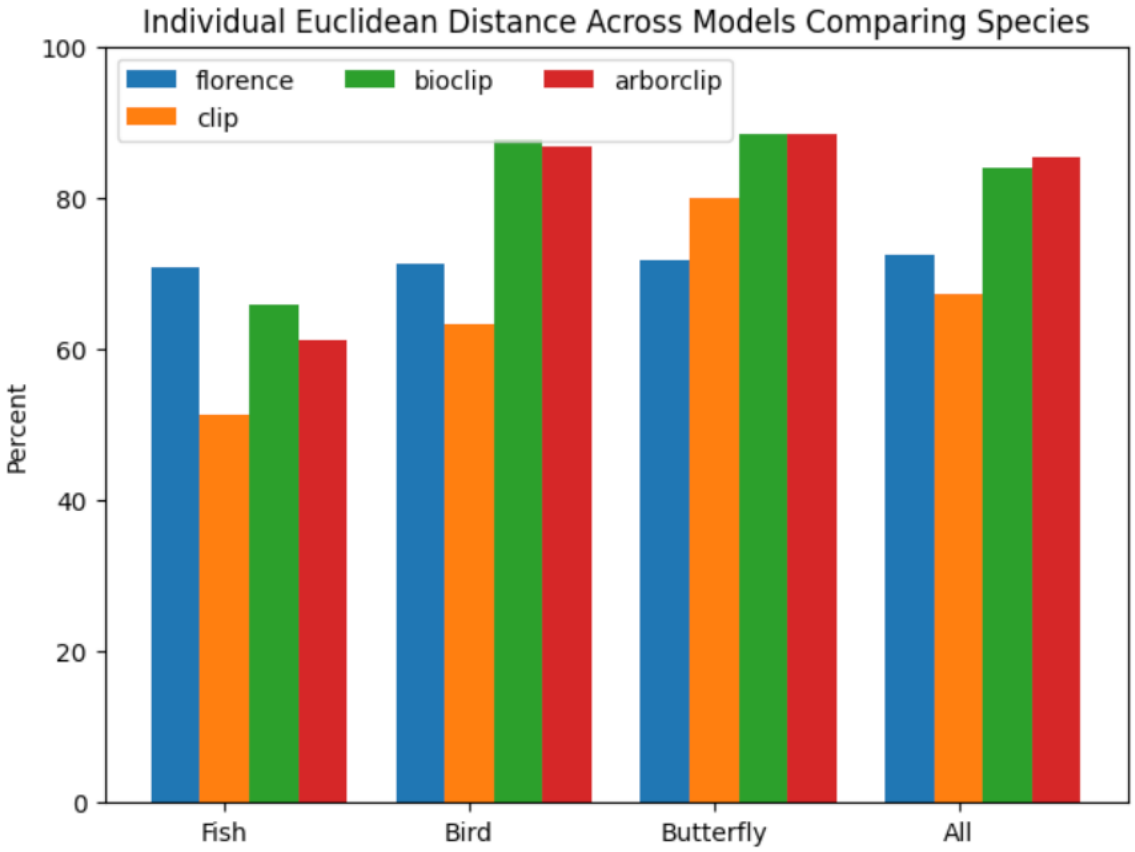
Results in comparing time to generate embeddings for 300 images for each model:

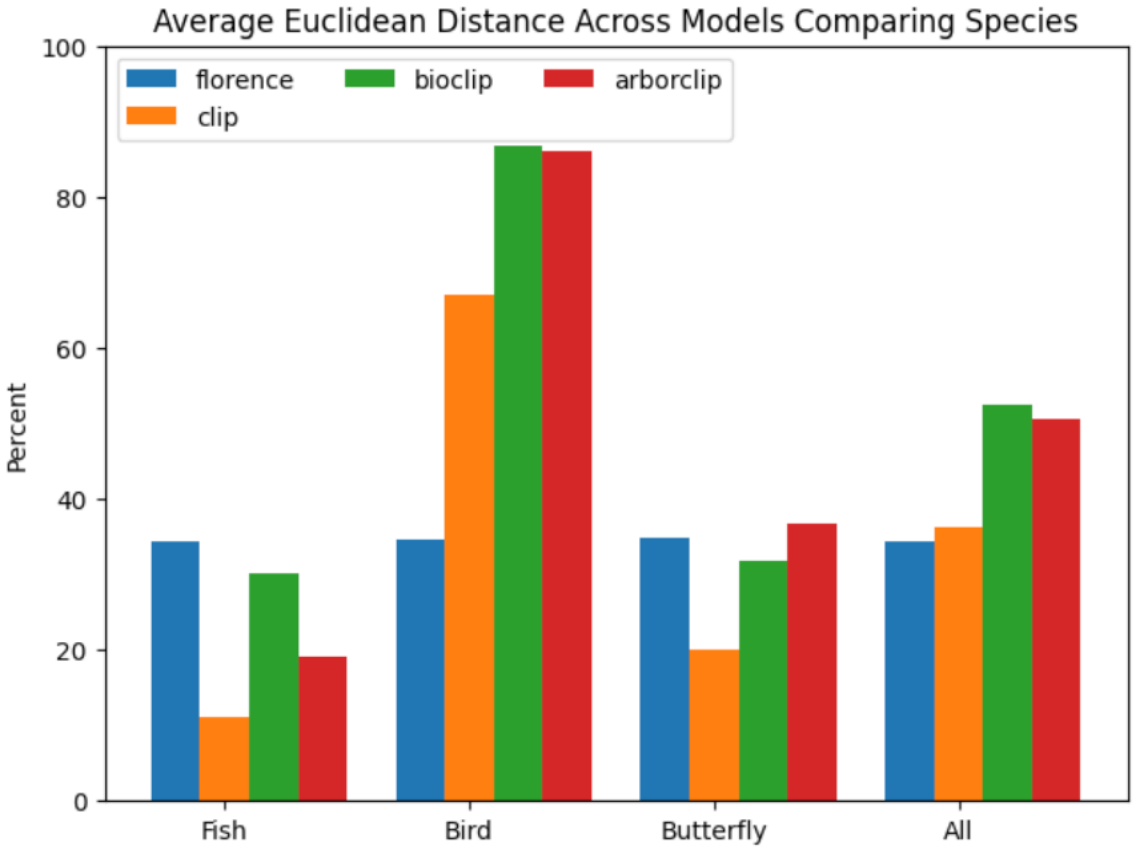


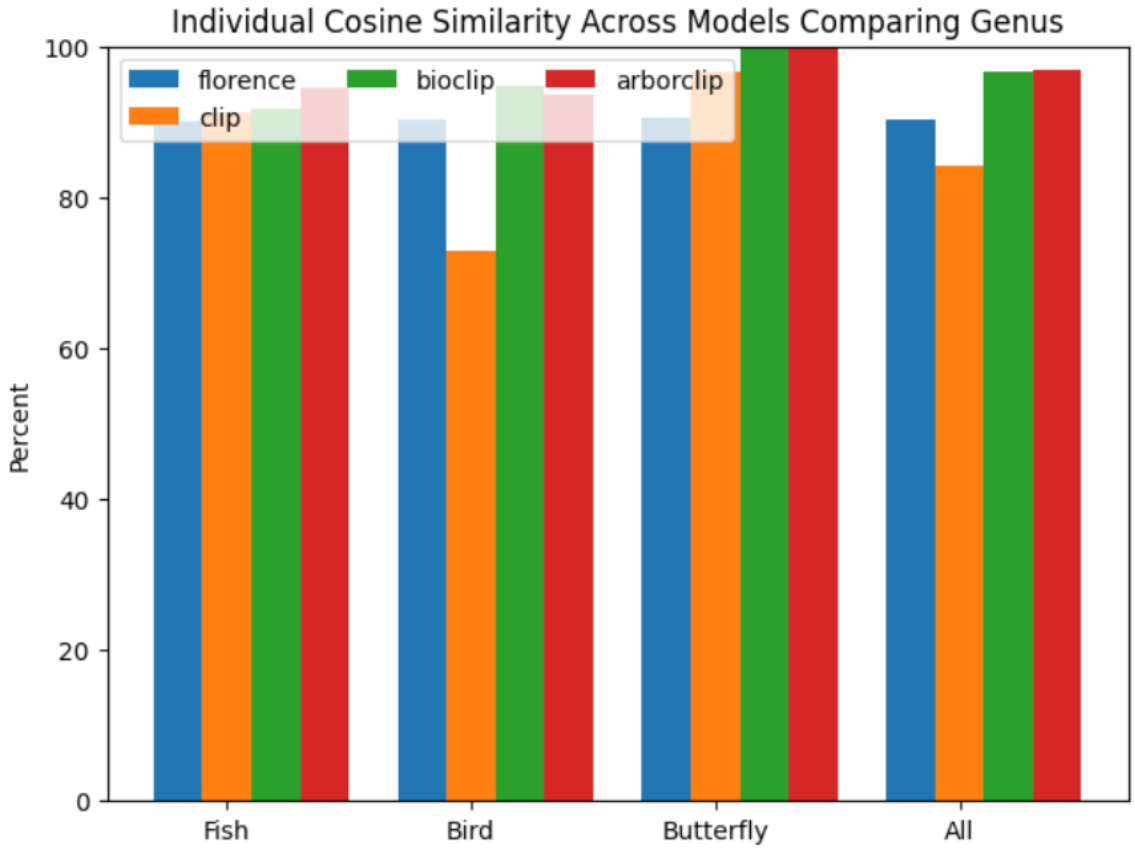
Graphs comparing Cosine Similarity and Euclidean Distance across Species and Genus:

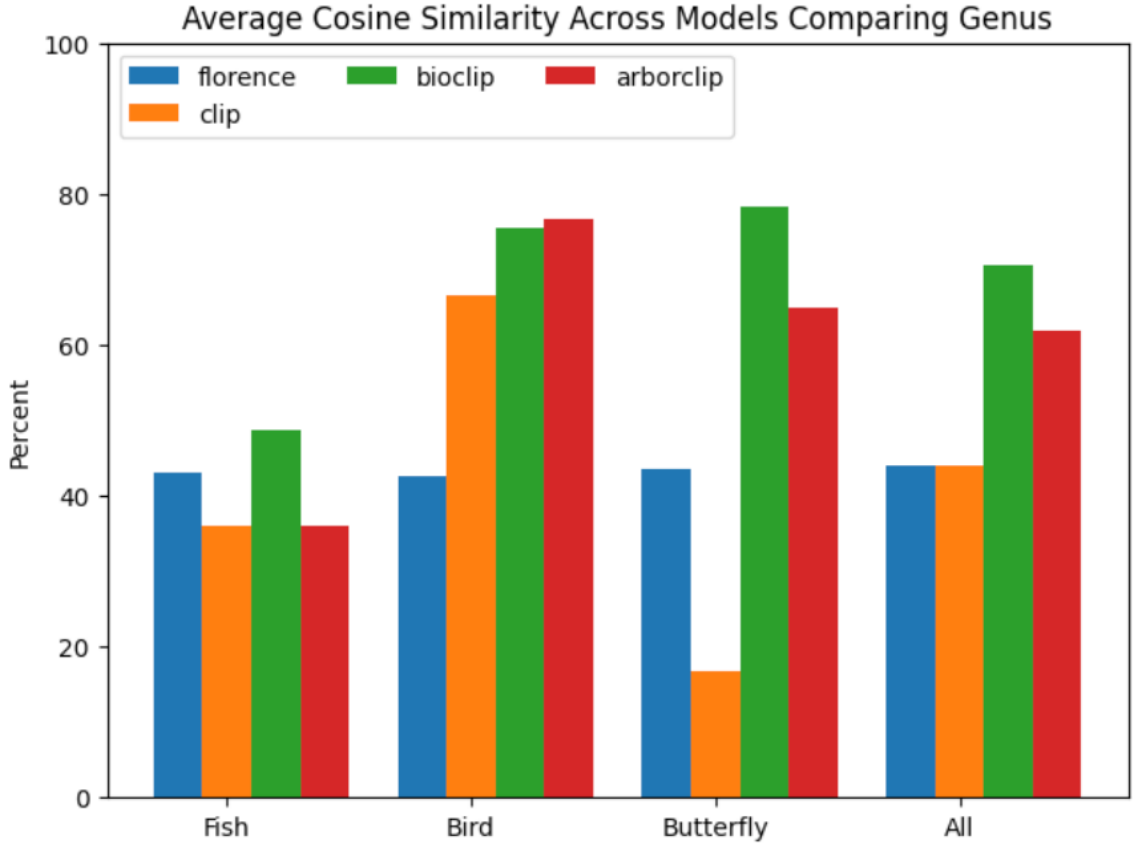

Individual Cosine Similarity Across Models Comparing Species
florence
clip
bioclip
arborclip
Percent
100
80
60
40
20
0
Fish
Bird
Butterfly
All

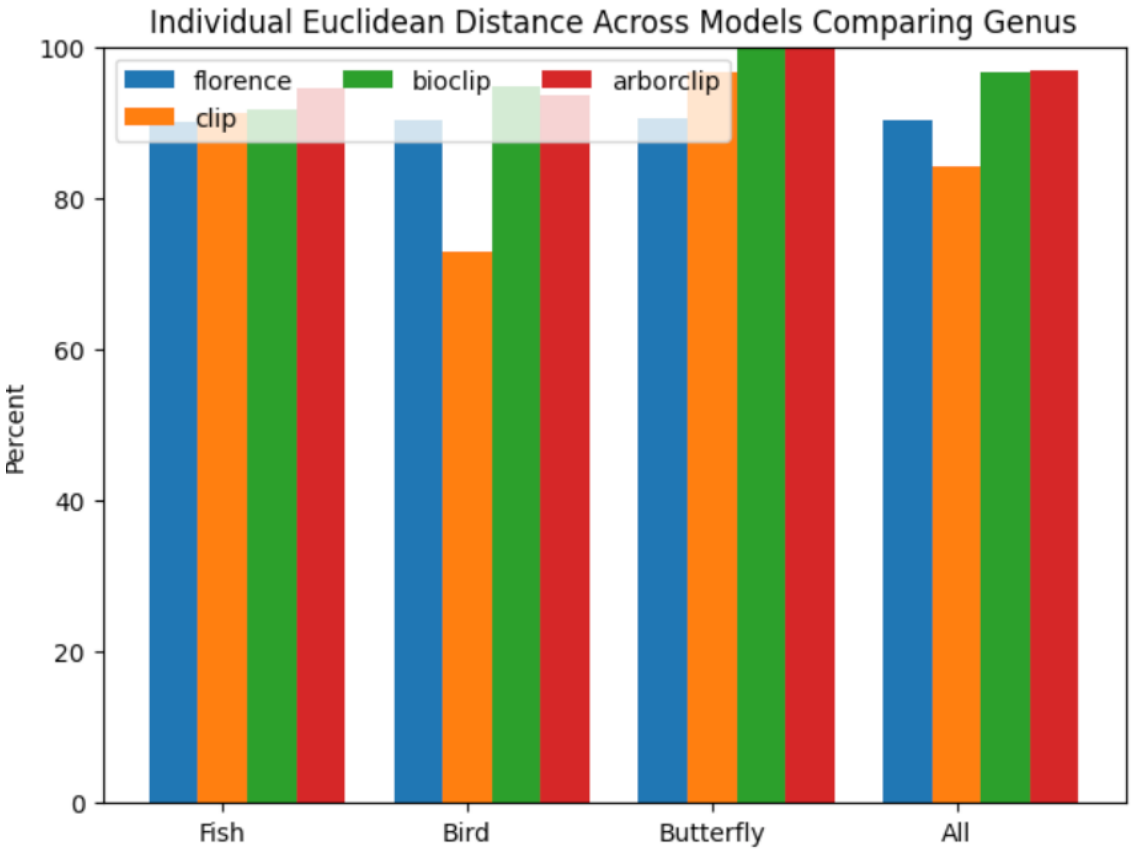

Average Cosine Similarity Across Models Comparing Species
florence
clip
bioclip
arborclip
Percent
100
80
60
40
20
0
Fish
Bird
Butterfly
All

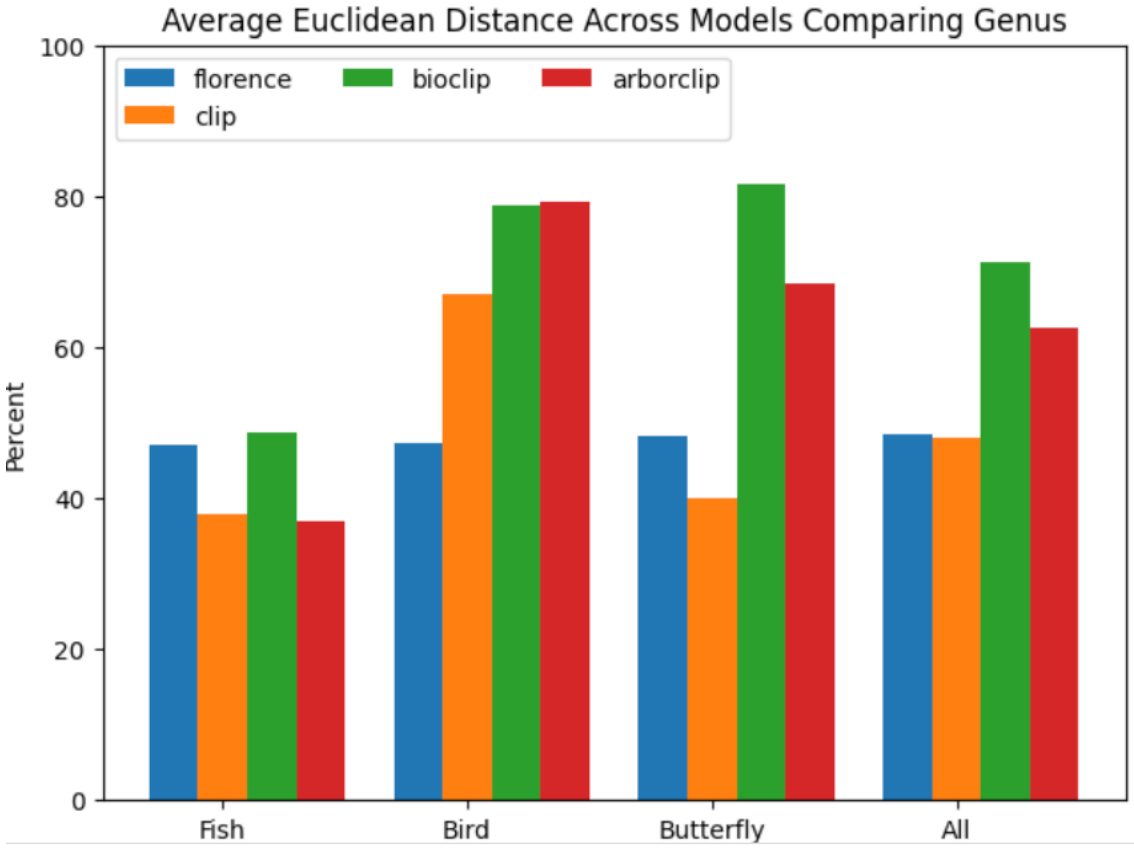

Individual Euclidean Distance Across Models Comparing Species
florence
clip
bioclip
arborclip
Percent
100
80
60
40
20
0
Fish
Bird
Butterfly
All


Average Euclidean Distance Across Models Comparing Species
florence
clip
bioclip
arborclip
Percent
100
80
60
40
20
0
Fish
Bird
Butterfly
All


Individual Cosine Similarity Across Models Comparing Genus
florence
clip
bioclip
arborclip
Percent
0
20
40
60
80
100
Fish
Bird
Butterfly
All


Average Cosine Similarity Across Models Comparing Genus
florence
clip
bioclip
arborclip
Percent
0
20
40
60
80
100
Fish
Bird
Butterfly
All


Individual Euclidean Distance Across Models Comparing Genus
florence
clip
bioclip
arborclip
Percent
100
80
60
40
20
0
Fish
Bird
Butterfly
All


Average Euclidean Distance Across Models Comparing Genus
florence
clip
bioclip
arborclip
Percent
100
80
60
40
20
0
Fish
Bird
Butterfly
All

5. Next Week Proposal

- Set up Ollama LLM viewer locally
- Test InternVL vision model
- Test implementation of Bioclip/Arborclip with Biocosmos
- Update Vue to use Vite and .vue components