Week 13 Report - NFHM

Romouald Dombrovski

November 15, 2024

1 Time Log

What progress did you make in the last week?

- Generated captions with InternVl using Thomas' methods
- Set up and familiarized myself with PACE ICE cluster
- Met with Thomas, Dr. Porto, Moritz and Bree to discuss training models with UNI-COM
- Refactored code into usable utils chunks
- Created training plan with Thomas for cluster comparison
- Trained cluster models with 100, 500 clusters
- Evaluated the models
- Met with Thomas and Bree to discuss results and next steps

2 Abstract

Bjerge K., et al. Hierarchical Classification of Insects with Multitask Learning and Anomaly Detection. doi: https://doi.org/10.1101/2023.06.29.546989.

Abstract Cameras and computer vision are revolutionising the study of insects, creating new research opportunities within agriculture, epidemiology, evolution, ecology and monitoring of biodiversity. However, a major challenge is the diversity of insects and close resemblances of many species combined with computer vision are often not sufficient to classify large numbers of insect species, which sometimes cannot be identified at the species level. Here, we present an algorithm to hierarchically classify insects from images, leveraging a simple taxonomy to (1) classify specimens across multiple taxonomic ranks simultaneously, and (2) highlight the lowest rank at which a reliable classification can be reached. Specifically, we propose multitask learning, a loss function incorporating class dependency at each taxonomic rank, and anomaly detection based on outlier analysis for quantification of uncertainty. First, we compile a dataset of 41,731 images of insects, combining images from time-lapse monitoring of floral scenes with images from the Global Biodiversity Information Facility (GBIF). Second, we adapt state-of-the-art convolutional neural networks, ResNet and EfficientNet, for the hierarchical classification of insects belonging to three orders, five families and nine species. Third, we assess model generalization for 11 species unseen by the trained models. Here, anomaly detection is used to predict the higher rank of the species not present in the training set. We found that incorporating a simple taxonomy into our model increased accuracy at higher taxonomic ranks. As expected, our algorithm correctly classified new insect species at higher taxonomic ranks, while classification was uncertain at lower taxonomic ranks. Anomaly detection can effectively flag novel taxa that are visually distinct from species in the training data. However, five novel taxa were consistently mistaken for visually similar species in the training data. Above all, we have demonstrated a practical approach to hierarchical classification based on species taxonomy and uncertainty during automated in situ monitoring of live insects. Our method is simple and versatile and could be implemented to classify a wide range of insects as well as other organisms.

Summary (GPT-4o) This paper introduces a hierarchical classification method for identifying insects from images, leveraging multitask learning, hierarchical loss functions, and anomaly detection to classify insects across multiple taxonomic ranks and flag uncertain classifications. Using a dataset of 41,731 insect images, the authors trained modified convolutional neural networks, such as ResNet50 and EfficientNetB3, to classify insects at the order, family, and species levels. The model demonstrated improved accuracy at higher taxonomic ranks and was particularly effective at classifying new, unseen species at broader taxonomic categories. Anomaly detection flagged species not present in the training data as "unsure," ensuring reliable higher-level classifications while highlighting uncertainty at lower ranks. By incorporating data from time-lapse monitoring and the Global Biodiversity Information Facility (GBIF), the model became robust across diverse image types. This approach holds promise for applications in biodiversity monitoring, pest management, and ecological studies, offering scalable and reliable classification systems for insect identification and novel species detection.

3 Scripts and Code Blocks

From last week we decided to continue working on training UNICOM models on different cluster sizes. I discussed with Thomas to ensure that we stay on the same page in terms of hyperparameters. We used OpenClip's ViT-H-14-378-quickgelu model to create initial embeddings for images and text. We used this model, because it had the best initial performance of any other CLIP model as per the accuracy comparison provided in the OpenCLIP repo. The model we trained was a CLIP ViT-B/32 visual transformer

The hyperparameters we decided to match were the following:

```
        self.batch_size = 128
        self.dataset = "cub"
        self.debug = 0
        self.epochs = 32
        self.lr = 1e-5
        self.lr_pfc_weight = 10.0
        self.input_size = 224
        self.gradient_acc = 1
        self.model_name = "ViT-B/32"
        self.margin_loss_m1 = 1.0
        self.margin_loss_m2 = 0.3
        self.margin_loss_m3 = 0.0
        self.margin_loss_s = 32.0
        self.margin_loss_filter = 0.0
        self.num_workers = 4
        self.num_feat = 512
        self.optimizer = "adamw"
        self.output = "/tmp/tmp_for_training"
        self.resume = "NULL"
        self.sample_rate = 1.0
        self.seed = 5
        self.transform = None
        self.weight_decay = 0
        self.output_dim = 512 if "B/32" in self.model_name else (
            768 if "B/16" in self.model_name else 1024
        )
```

This was the main loop calling each of step of the evaluation

```
eval_df = pd.DataFrame(columns=['model', 'taxa', 'column',
    'accuracy_individual', 'accuracy_avg', 'distance_individual', 'distance_avg'])

for ncluster in [100, 500, 1000, 2000]:

    # create cluster train df
    train_df = setup_clusters(other_df, ncluster)
    config = Config()
    model, transform = training_loop(config, train_df)
    eval_row = vlm_species_eval(other_df, model, transform, get_embeds, ncluster)
    torch.save(model.state_dict(),
        f'/home/hice1/rdombrovski3/scratch/unicom_clip_weights_{ncluster}_clusters.pth')
    del model
    torch.cuda.empty_cache()
    eval_df = pd.concat([eval_df, eval_row], ignore_index=True)
```

The training loop was the same as the previous week, so I won't repeat it here. After discussion with the team, I wanted to compare evaluation by doing the initial clustering + training on the ENTIRE VLM4Bio dataset, instead of doing the initial split. Whilst there was risk of overfitting when doing the evaluation, we wanted to see if it greatly affected the results (the clustering process was already a regularizing method). Here is how the clusters were set up for my process:

```
def setup_clusters(df, num_cluster):

    train_df, test_df = get_test_train_split(df, test_size=0,
        stratify_by_scientific_name=True)

    # make sure clusters are set on training data ONLY
    train_df = set_clusters_on_df(train_df, 'avg_embedding',
        ncentroids=num_cluster)
    return train_df
```

I rewrote a lot of my code from notebooks into reusable utils functions. The code can be seen in the following pull request.

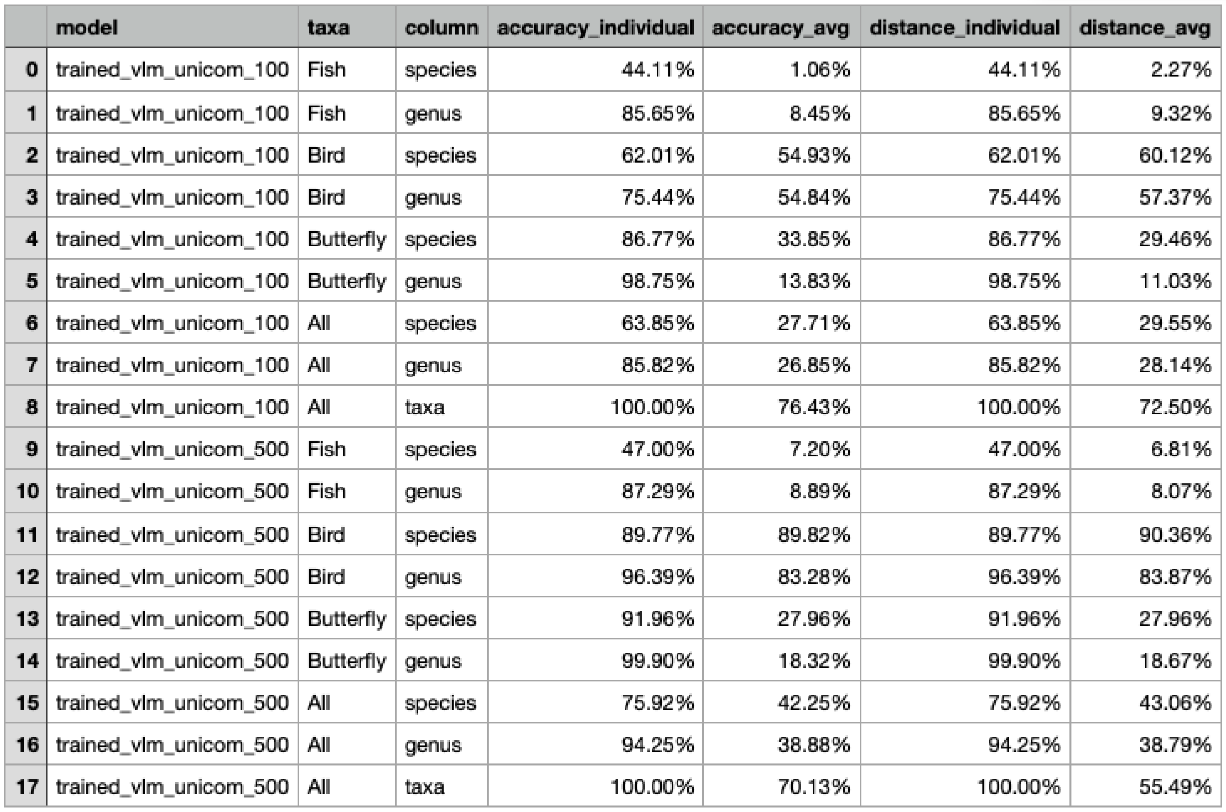
Here is an example, which is the rewritten metrics evaluation function

```
import os
import pandas as pd
import requests
import torch
from transformers import AutoModelForCausalLM, AutoProcessor, AutoModel,
    CLIPImageProcessor
from PIL import Image
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity, euclidean_distances
from sklearn.model_selection import train_test_split
import open_clip
from huggingface_hub import hf_hub_download
from time import time
```

```
device = "cuda" if torch.cuda.is_available() else "cpu"
torch_dtype = torch.float16 if torch.cuda.is_available() else torch.float32

def get_test_train_split(df_cleaned, test_size=0.2,
    stratify_by_scientific_name=False):
    """
    Split a dataframe into test and train sets
    Parameters
    ----------
    df_cleaned : dataframe
        DESCRIPTION.
    test_size : float, optional
        size of the test set. The default is 0.1.
    stratify_by_scientific_name : Boolean, optional
        Allow for stratification (even representation across classes). The
            default is False.
    Returns
    -------
    train_df : TYPE
        DESCRIPTION.
    test_df : TYPE
        DESCRIPTION.
    """
    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['scientific_name'].value_counts()
        valid_species = species_counts[species_counts > 1].index
        df_filtered =
            df_cleaned[df_cleaned['scientific_name'].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['scientific_name'])
    else:
        train_df, test_df = train_test_split(df_cleaned, test_size=test_size)
        return train_df, test_df


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

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_avg_sim_labels,highest_avg_

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_score,
        lowest_avg_dist_name, lowest_avg_dist_score = find_best_match_vectorized(
        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_individual * 100:.2f}%")
    print(f"Accuracy for model {model} on column {label_column} based on highest
        mean cosine similarity: {accuracy_mean * 100:.2f}%")
    print(f"Accuracy for model {model} on column {label_column} based on lowest
        individual euclidean distance: {accuracy_individual * 100:.2f}%")
    print(f"Accuracy for model {model} on column {label_column} based on lowest
        mean euclidean distance: {accuracy_mean * 100:.2f}%")

    return test_df, accuracy_individual, accuracy_mean, accuracy_dist,
        accuracy_avg_dist

def extract_genus(scientific_name):

    try:
        return scientific_name.split()[0]
    except Exception as e:
        print(scientific_name)
        raise

def vlm_species_eval(full_species_df, model, transform, get_embeds):
    """
    Evaluation loop to calculate cosine similarity and euclidean distance in n=1
        NN method
    Parameters
    ----------
    full_species_df : Dataframe
        Dataframe containing all info about the VLM4Bio datasets, with image file
            names and scientific_names.
    model : model
        model used to evaluate the data.
    transform : function
        transform function used to preprocess data for the model.
    get_embeds : function
        function used to get image embeddings for the model.
    Returns
    -------
    eval_df : Dataframe
        output dataframe including all the accuracy calculations.
    """
    full_species_df['category'] = full_species_df['category'].apply(lambda x:
        x.strip())

    eval_df = pd.DataFrame(columns=['model', 'taxa', 'column',
        'accuracy_individual', 'accuracy_avg', 'distance_individual',
        'distance_avg'])
    rows = []
    model = model.float()
    analysis_taxa = ['Fish', 'Bird', 'Butterfly', 'All']

    full_species_df =
        full_species_df[full_species_df['scientific_name'].str.strip() != '']
    full_species_df['genus'] =
        full_species_df['scientific_name'].apply(extract_genus)

    # note the amount of time necessary to generate all embeddings
    start = time.time()

    full_species_df['image_embeddings'] =
        full_species_df['image_name'].apply(lambda x: get_embeds(x, model,
        transform))
    full_species_df.dropna(subset=['scientific_name'], inplace=True)

    time_length = time.time() - start
    print(f'Took {time_length} seconds to generate embeddings')

    # we're gonna compare species by species within each taxa, and against each
    for taxa in analysis_taxa: # cleaned_dfs can have full species appended to it
        too
        print(f"WORKING ON TAXA: {taxa}")

        df = full_species_df if taxa == 'All' else
            full_species_df[full_species_df['category'] == taxa]
        train_df, test_df = get_test_train_split(df, test_size=0.2,
            stratify_by_scientific_name=False)

        start = time.time()
        test_df, accuracy_individual, accuracy_avg, distance_individual,
            distance_avg = apply_best_match_vectorized(test_df, train_df,
            'scientific_name', 'trained_vlm_unicom')
        time_length = time.time() - start
        print(f"Time for analysis for model trained_vlm_unicom: {time_length}")
        rows.append({'model': 'trained_vlm_unicom', 'taxa': taxa, 'column':
            'species', 'accuracy_individual': f"{accuracy_individual * 100:.2f}%",
            'accuracy_avg': f"{accuracy_avg * 100:.2f}%", 'distance_individual':
            f"{distance_individual * 100:.2f}%", 'distance_avg': f"{distance_avg *
            100:.2f}%"})

        test_df, accuracy_individual, accuracy_avg, distance_individual,
            distance_avg = apply_best_match_vectorized(test_df, train_df, 'genus',
            'trained_vlm_unicom')
        rows.append({'model': 'trained_vlm_unicom', 'taxa': taxa, 'column':
            'genus', 'accuracy_individual': f"{accuracy_individual * 100:.2f}%",
            'accuracy_avg': f"{accuracy_avg * 100:.2f}%", 'distance_individual':
            f"{distance_individual * 100:.2f}%", 'distance_avg': f"{distance_avg *
            100:.2f}%"})
        if taxa == 'All': # only relevant when comparing against other taxa
            test_df, accuracy_individual, accuracy_avg, distance_individual,
                distance_avg = apply_best_match_vectorized(test_df, train_df,
                'category', 'trained_vlm_unicom')
            rows.append({'model': 'trained_vlm_unicom', 'taxa': taxa, 'column':
                'taxa', 'accuracy_individual': f"{accuracy_individual *
                100:.2f}%", 'accuracy_avg': f"{accuracy_avg * 100:.2f}%",
                'distance_individual': f"{distance_individual * 100:.2f}%",
                'distance_avg': f"{distance_avg * 100:.2f}%"})

    eval_df = pd.concat([eval_df, pd.DataFrame(rows)], ignore_index=True)
    return eval_df
```

4 Visualization

I had some difficulties with the PACE ICE cluster, especially concerning length of training time + memory usage, as this was my first time using the tool. Because of this, I was only able to train a model on 100 500 clusters. I will be running the training loop with 1000 and 2000 clusters in the coming day(s) and compare the evaluation to Bioclip and OpenClip models.



	model	taxa	column	accuracy_individual	accuracy_avg	distance_individual	distance_avg
0	trained_vlm_unicom_100	Fish	species	44.11%	1.06%	44.11%	2.27%
1	trained_vlm_unicom_100	Fish	genus	85.65%	8.45%	85.65%	9.32%
2	trained_vlm_unicom_100	Bird	species	62.01%	54.93%	62.01%	60.12%
3	trained_vlm_unicom_100	Bird	genus	75.44%	54.84%	75.44%	57.37%
4	trained_vlm_unicom_100	Butterfly	species	86.77%	33.85%	86.77%	29.46%
5	trained_vlm_unicom_100	Butterfly	genus	98.75%	13.83%	98.75%	11.03%
6	trained_vlm_unicom_100	All	species	63.85%	27.71%	63.85%	29.55%
7	trained_vlm_unicom_100	All	genus	85.82%	26.85%	85.82%	28.14%
8	trained_vlm_unicom_100	All	taxa	100.00%	76.43%	100.00%	72.50%
9	trained_vlm_unicom_500	Fish	species	47.00%	7.20%	47.00%	6.81%
10	trained_vlm_unicom_500	Fish	genus	87.29%	8.89%	87.29%	8.07%
11	trained_vlm_unicom_500	Bird	species	89.77%	89.82%	89.77%	90.36%
12	trained_vlm_unicom_500	Bird	genus	96.39%	83.28%	96.39%	83.87%
13	trained_vlm_unicom_500	Butterfly	species	91.96%	27.96%	91.96%	27.96%
14	trained_vlm_unicom_500	Butterfly	genus	99.90%	18.32%	99.90%	18.67%
15	trained_vlm_unicom_500	All	species	75.92%	42.25%	75.92%	43.06%
16	trained_vlm_unicom_500	All	genus	94.25%	38.88%	94.25%	38.79%
17	trained_vlm_unicom_500	All	taxa	100.00%	70.13%	100.00%	55.49%

Figure 1: Table showing comparison between models trained on 100 500 clusters

5 Next Week Proposal

- Finish running training + eval on remaining clusters
- Get set up on Hyergator
- Test parallel GPU computing for UNICOM training
- Re-read and discuss Bioclip paper with Thomas
- Work with Thomas on getting evaluation scripts aligned
- Meet with Dr. Porto and UofF collaboration team