Week 16 Report - NFHM

Romouald Dombrovski

December 6, 2024

1 Time Log

What progress did you make in the last week?

- Prepared slides with Thomas for HAAG presentation
- Prepared graphs for Dr. Porto's presentation
- Presented our research to the rest of the HAAG groups
- Updated + Ran SLURM script to download rough ToL datasets
- Updated + Ran SLURM script to generate webdataset files
- Updated + Ran SLURM script to check for bad shards

2 Abstract

Jonathan A. Rees and Karen Cranston. Automated assembly of a reference taxonomy for phylogenetic data synthesis. Biodiversity Data Journal, 5:e12581, 2017.

Abstract Taxonomy and nomenclature data are critical for any project that synthesizes biodiversity data, as most biodiversity data sets use taxonomic names to identify taxa. Open Tree of Life is one such project, synthesizing sets of published phylogenetic trees into comprehensive summary trees. No single published taxonomy met the taxonomic and nomenclatural needs of the project. Here we describe a system for reproducibly combining several source taxonomies into a synthetic taxonomy, and we discuss the challenges of taxonomic and nomenclatural synthesis for downstream biodiversity projects.

Summary (GPT-4o) The paper Automated assembly of a reference taxonomy for phylogenetic data synthesis by Rees and Cranston outlines the development of the Open Tree Taxonomy (OTT), a system designed to synthesize multiple existing taxonomic databases into a comprehensive and reproducible reference taxonomy for the Open Tree of Life project. This taxonomy supports phylogenetic data synthesis by combining ten diverse sources, each contributing unique taxonomic data, and resolving conflicts using a priority-based, automated assembly process. The methodology employs heuristics to align and merge taxa while maintaining provenance for transparency. It addresses challenges such as synonymies, homonymies, and inconsistent taxonomic ranks across sources, integrating curator-driven adjustments and feedback mechanisms to improve accuracy. The resulting taxonomy, comprising millions of nodes and tips, enables enhanced phylogenetic and taxonomic representation, supports ongoing updates, and remains openly accessible, aligning with the broader goals of reproducible and collaborative biodiversity research.

3 Scripts and Code Blocks

I followed the steps described in Bioclip's documentation on 'How to Create TreeOfLife - 10M' as found in https://github.com/Imageomics/bioclip/blob/main/docs/imageomics/treeoflife10m.md The scripts were mostly easy to follow, but I had to make sure that the correct paths were being accessed when referring to our group storage. For reference, we are running our scripts on the University of Florida's Hipergator cluster, which gives 30Gb of personal storage for user, and 20Tb storage per group. As this dataset consisted of 10M images, we definitely required the use of the group storage. The set up script (containing relevant bash functions and variables) looked like this:

```
#!/bin/bash

REPO_ROOT="$PWD"
SLURM_SUBMIT_DIR="$REPO_ROOT/slurm"

export REPO_ROOT SLURM_SUBMIT_DIR

# IF YOU WOULD LIKE TO SET A CUSTOM DATASET PATH OR SLURM DIRECTIVES, CHANGE THE
    VARIABLES BELOW


######################################## customize-below >>>


# optional:
# Set a custom path for the dataset.
# It defaults to the location below within the git repository.
# If you leave the default location, ensure you have placed the repository in a
    location with sufficient space.
dataset_path="/blue/arthur.porto-biocosmos/rdombrovski3.gatech/data/TreeOfLife-10M"

SLURM_NODES=1

# SLURM directives for each job
# Wall times are set conservatively to ensure jobs complete on most systems

# Quick download of metadata files
METADATA_SLURM_TIME="0:20:00"
METADATA_SLURM_NTASKS_PER_NODE=1
METADATA_SLURM_OUTPUT="${REPO_ROOT}/logs/download_metadata_%j.out"
METADATA_SLURM_ERROR="${REPO_ROOT}/logs/download_metadata_%j.err"
METADATA_PARTITION="hpg-dev"

# Network-intenstive download of 63x ~30GB .tar.gz files with EOL images (~2.5h
    on OSC, 200-300MiB/s)
```

```
# Wall time may need to increase if ‘wget‘ is used instead of ‘aria2c‘
EOL_SLURM_TIME="18:00:00"
EOL_SLURM_NTASKS_PER_NODE=12
EOL_SLURM_OUTPUT="${REPO_ROOT}/logs/download_eol_%j.out"
EOL_SLURM_ERROR="${REPO_ROOT}/logs/download_eol_%j.err"
EOL_PARTITION="hpg-dev"

# Network intensive download (~11min on OSC, 340MiB/s), CPU-bound extraction
    (~88min total on OSC)
INAT21_SLURM_TIME="4:00:00"
INAT21_SLURM_NTASKS_PER_NODE=1
INAT21_SLURM_OUTPUT="${REPO_ROOT}/logs/download_inat21_%j.out"
INAT21_SLURM_ERROR="${REPO_ROOT}/logs/download_inat21_%j.err"
INAT21_PARTITION="hpg-dev"

# Modest download, CPU-bound extraction (made fast by parallelization) (~8min on
    OSC)
BIOSCAN_SLURM_TIME="2:00:00"
BIOSCAN_SLURM_NTASKS_PER_NODE=20
BIOSCAN_SLURM_OUTPUT="${REPO_ROOT}/logs/download_bioscan_%j.out"
BIOSCAN_SLURM_ERROR="${REPO_ROOT}/logs/download_bioscan_%j.err"
BIOSCAN_PARTITION="hpg-dev"

######################################## customize-above <<<

export SLURM_ACCOUNT SLURM_NODES
export METADATA_SLURM_TIME METADATA_SLURM_NTASKS_PER_NODE METADATA_SLURM_OUTPUT
    METADATA_SLURM_ERROR
export EOL_SLURM_TIME EOL_SLURM_NTASKS_PER_NODE EOL_SLURM_OUTPUT EOL_SLURM_ERROR
export INAT21_SLURM_TIME INAT21_SLURM_NTASKS_PER_NODE INAT21_SLURM_OUTPUT
    INAT21_SLURM_ERROR
export BIOSCAN_SLURM_TIME BIOSCAN_SLURM_NTASKS_PER_NODE BIOSCAN_SLURM_OUTPUT
    BIOSCAN_SLURM_ERROR

metadata_path="${dataset_path}/metadata"
names_path="${metadata_path}/naming"
eol_root="${dataset_path}/dataset/eol"
inat21_root="${dataset_path}/dataset/inat21"
bioscan_root="${dataset_path}/dataset/bioscan"
rarespecies_root="${dataset_path}/dataset/rarespecies"
container_path="${REPO_ROOT}/containers"
logs_path="${REPO_ROOT}/logs"

export dataset_path metadata_path names_path eol_root inat21_root bioscan_root
    rarespecies_root container_path logs_path
```

```
# Create directories
mkdir -p "${dataset_path}" "${metadata_path}" "${names_path}" "${eol_root}"
    "${inat21_root}" "${bioscan_root}" "${rarespecies_root}" "${container_path}"
    "${logs_path}"

is_slurm_job() {
    [ -n "$SLURM_JOB_ID" ]
}

# Download function for fast download with aria2c if Apptainer is available,
    otherwise uses wget
download() {
    echo "Downloading $1 to $2/$3"
    url=$1
    output_path=$2
    output_file=$3

    wget -c -O "${output_path}/${output_file}" "$url"
}

export -f download

# Function to extract BIOSCAN-1M
extract_bioscan() {
    local zip_file="$1"
    local output_dir="$2"
    local temp_dir="${output_dir}/temp"
    local final_file_count=1128313

    mkdir -p "$temp_dir"

    # Adapt resource usage based on environment being used
    local available_cores=$(nproc)
    if is_slurm_job; then
        # Use at least 1
        local requested_tasks=${BIOSCAN_SLURM_NTASKS_PER_NODE:-1}
        requested_tasks=$(( requested_tasks < 1 ? 1 : requested_tasks ))

        # Use the smaller of number of cores requested vs available
        local num_tasks=$(( requested_tasks < available_cores ? requested_tasks :
            available_cores ))
    else
        # For command line, use all available cores up to a maximum of 4
        local num_tasks=$(( available_cores < 4 ? available_cores : 4 ))
    fi
```

```
    # Extract all parts in parallel
    total_parts=$(unzip -Z1 "$zip_file" | grep 'bioscan/images/cropped_256/part'
        | cut -d'/' -f4 | sort -u | wc -l)
    current_part=0
    # Prevent race to create temp parent directories during unzip
    mkdir -p "$temp_dir/bioscan/images/cropped_256/"
    unzip -Z1 "$zip_file" | grep 'bioscan/images/cropped_256/part' | cut -d'/'
        -f4 | sort -u | \
        xargs -P "$num_tasks" -I {} sh -c '
            unzip -q "$0" "bioscan/images/cropped_256/{}/*" -d "$1"
            echo "Extracted part {}"
        ' "$zip_file" "$temp_dir"

    # Move all part directories to the final location
    find "$temp_dir/bioscan/images/cropped_256/" -type d -name 'part*' -print0 |
        xargs -0 -I {} mv {} "$output_dir/"

    # Remove the temp directory
    rm -rf "$temp_dir"

    # Count the number of extracted files and verify
    local extracted_file_count
    extracted_file_count=$(find "$output_dir" -type f -name '*.jpg' | wc -l)

    if [ "$extracted_file_count" -ne "$final_file_count" ]; then
        echo "Error: Expected $final_file_count files, but found
            $extracted_file_count files." >&2
        return 1
    fi

    rm "$zip_file"

    echo "Extracted BIOSCAN-1M to $output_dir"
}

export -f extract_bioscan

# Function to extract iNat21
extract_inat21() {
    local tar_archive="$1"
    local output_dir="$2"

    tar -xzf "$tar_archive" -C "$output_dir"

    rm "$tar_archive"
    echo "Extracted iNat21 to $output_dir"
```

```
}

export -f extract_inat21

# Container setup for aria2c if Apptainer is available
aria2_tag="202209060423"
aria2_sif_path="${container_path}/aria2-pro_${aria2_tag}.sif"

if command -v apptainer &> /dev/null; then
    mkdir -p "$container_path"
    aria2_sif_path=""
fi
```

The set up was used when running the submit script with the command: ‘sbatch scripts/-submit_download_tol-10m_components.bash‘

```
#!/bin/bash

set -ex

# Run from the root of the repository.
# Usage: sbatch --account <your-account>
    scripts/submit_download_tol-10m_components.bash

# Source the setup script
source "scripts/setup_download_tol-10m_components.bash"
export SBATCH_ACCOUNT=$SLURM_JOB_ACCOUNT # Applies to all child jobs

# Ensure necessary directories exist
mkdir -p "$logs_path"

# Submit jobs
metadata_job=$(sbatch \
    --time=$METADATA_SLURM_TIME \
    --nodes=$SLURM_NODES \
    --ntasks-per-node=$METADATA_SLURM_NTASKS_PER_NODE \
    --output=$METADATA_SLURM_OUTPUT \
    --error=$METADATA_SLURM_ERROR \
    --partition=$METADATA_PARTITION \
    "$SLURM_SUBMIT_DIR/download_metadata.slurm" | awk '{print $4}')

eol_job=$(sbatch \
    --time=$EOL_SLURM_TIME \
    --nodes=$SLURM_NODES \
    --ntasks-per-node=$EOL_SLURM_NTASKS_PER_NODE \
    --output=$EOL_SLURM_OUTPUT \
    --error=$EOL_SLURM_ERROR \
```

```
    --partition=$EOL_PARTITION \
    "$SLURM_SUBMIT_DIR/download_eol.slurm" | awk '{print $4}')

inat21_job=$(sbatch \
    --time=$INAT21_SLURM_TIME \
    --nodes=$SLURM_NODES \
    --ntasks-per-node=$INAT21_SLURM_NTASKS_PER_NODE \
    --output=$INAT21_SLURM_OUTPUT \
    --error=$INAT21_SLURM_ERROR \
    --partition=$INAT21_PARTITION \
    "$SLURM_SUBMIT_DIR/download_inat21.slurm" | awk '{print $4}')

bioscan_job=$(sbatch \
    --time=$BIOSCAN_SLURM_TIME \
    --nodes=$SLURM_NODES \
    --ntasks-per-node=$BIOSCAN_SLURM_NTASKS_PER_NODE \
    --output=$BIOSCAN_SLURM_OUTPUT \
    --error=$BIOSCAN_SLURM_ERROR \
    --partition=$BIOSCAN_PARTITION \
    "$SLURM_SUBMIT_DIR/download_bioscan.slurm" | awk '{print $4}')

echo "Submitted jobs: metadata ($metadata_job), EOL ($eol_job), iNat21
    ($inat21_job), BIOSCAN ($bioscan_job)"

echo "Repo root: $REPO_ROOT"
```

The above SLURM jobs ran for nearly a whole day, and completed by downloading all the relevant datasets that made up Tree-of-Life 10M. The datasets were iNat21, bioscan, and EOL. I then ran the script to generate webdataset files.
The following was a file consisting of constants used by the script:

```
"""
All filepaths.
"""

# Actual datasets
DATASET_DIR =
    "/blue/arthur.porto-biocosmos/rdombrovski3.gatech/data/TreeOfLife-10M/dataset/"

eol_root_dir = f"{DATASET_DIR}eol"
inat21_root_dir = f"{DATASET_DIR}inat21/train"
bioscan_root_dir = f"{DATASET_DIR}bioscan"

# rare species
seen_in_training_json = "data/rarespecies/seen_in_training.json"
unseen_in_training_json = "data/rarespecies/unseen_in_training.json"
```

```
# Files we make
METADATA_DIR = "data/TreeOfLife-10M/metadata/"

eol_name_lookup_json = f"{METADATA_DIR}naming/eol_name_lookup.json"
inat21_name_lookup_json = f"{METADATA_DIR}naming/inat21_name_lookup.json"
bioscan_name_lookup_json = f"{METADATA_DIR}naming/bioscan_name_lookup.json"

db = f"{METADATA_DIR}mapping.sqlite"
```

Which was used in the following script:

```
    """
Writes the training and validation data to webdataset format.
"""
import argparse
import collections
import json
import logging
import multiprocessing
import os
import tarfile

from PIL import Image, ImageFile

from imageomics import disk_reproduce, eol_reproduce, evobio10m_reproduce,
    naming_reproduce, wds

########
# CONFIG
########

log_format = "[%(asctime)s] [%(levelname)s] [%(name)s] %(message)s"
logging.basicConfig(level=logging.INFO, format=log_format)
rootlogger = logging.getLogger("root")

Image.MAX_IMAGE_PIXELS = 30_000**2 # 30_000 pixels per side
ImageFile.LOAD_TRUNCATED_IMAGES = True


########
# SHARED
########


def load_img(file):
    img = Image.open(file)
    if img.mode in ("RGBA", "P"):
```

```
        img = img.convert("RGB")
    return img.resize(resize_size, resample=Image.BICUBIC)


def load_blacklists():
    image_blacklist = set()
    species_blacklist = set()

    with open(disk_reproduce.seen_in_training_json) as fd:
        for scientific, images in json.load(fd).items():
            image_blacklist |= set(os.path.basename(img) for img in images)

    with open(disk_reproduce.unseen_in_training_json) as fd:
        for scientific, images in json.load(fd).items():
            image_blacklist |= set(os.path.basename(img) for img in images)
            species_blacklist.add(scientific)

    return image_blacklist, species_blacklist


######################
# Encyclopedia of Life
######################


def copy_eol_from_tar(sink, imgset_path):
    logger = logging.getLogger(f"p{os.getpid()}")

    db = evobio10m_reproduce.get_db(db_path)
    select_stmt = "SELECT evobio10m_id, content_id, page_id FROM eol;"
    eol_ids_lookup = {
        evobio10m_id: (content_id, page_id)
        for evobio10m_id, content_id, page_id in
            db.execute(select_stmt).fetchall()
    }
    db.close()

    name_lookup =
        naming_reproduce.load_name_lookup(disk_reproduce.eol_name_lookup_json,
        keytype=int)

    # r|gz indcates reading from a gzipped file, streaming only
    with tarfile.open(imgset_path, "r|gz") as tar:
        for i, member in enumerate(tar):
            eol_img = eol_reproduce.ImageFilename.from_filename(member.name)
            if eol_img.raw in image_blacklist:
```

```
        continue

    # Match on treeoflife_id filename
    if eol_img.tol_id not in eol_ids_lookup:
        print(f"Can't find the tol_id {eol_img.tol_id}")
        logger.warning(
            "EvoBio10m ID missing. [tol_id: %s]",
            eol_img.tol_id,
        )
        continue

    # fetching page ID
    content_id, page_id = eol_ids_lookup[eol_img.tol_id]
    global_id = eol_img.tol_id

    # checking for global id in split
    if global_id not in splits[args.split] or global_id in finished_ids:
        continue

    # checking for page id
    if page_id not in name_lookup:
        continue

    # using name lookup for taxon, common, classes
    taxon, common, classes = name_lookup[page_id]

    if taxon.scientific in species_blacklist:
        continue

    file = tar.extractfile(member)
    try:
        img = load_img(file).resize(resize_size)
    except OSError as err:
        logger.warning(
            "Error opening file. Skipping. [tar: %s, err: %s]",
                imgset_path, err
        )
        continue

    txt_dct = make_txt(taxon, common)
    # writing EOL
    sink.write(
        {"__key__": global_id, "jpg": img, **txt_dct} #, "classes":
            classes} # REMOVED "classes", unused list of numbers, throws
            an error as an unknown extension with webdataset
    )
```

```
########
# INAT21
########


def copy_inat21_from_clsdir(sink, clsdir):
    logger = logging.getLogger(f"p{os.getpid()}")

    db = evobio10m_reproduce.get_db(db_path)
    select_stmt = "SELECT evobio10m_id, filename, cls_num FROM inat21;"
    evobio10m_id_lookup = {
        (filename, cls_num): evobio10m_id
        for evobio10m_id, filename, cls_num in db.execute(select_stmt).fetchall()
    }
    db.close()

    name_lookup =
        naming_reproduce.load_name_lookup(disk_reproduce.inat21_name_lookup_json,
        keytype=int)

    clsdir_path = os.path.join(disk_reproduce.inat21_root_dir, clsdir)
    for i, filename in enumerate(os.listdir(clsdir_path)):
        filepath = os.path.join(clsdir_path, filename)

        cls_num, *_ = clsdir.split("_")
        cls_num = int(cls_num)

        if (filename, cls_num) not in evobio10m_id_lookup:
            logger.warning(
                "Evobio10m ID missing. [image: %s, cls: %d]", filename, cls_num
            )
            continue

        global_id = evobio10m_id_lookup[(filename, cls_num)]
        if global_id not in splits[args.split] or global_id in finished_ids:
            continue

        taxon, common, classes = name_lookup[cls_num]

        if taxon.scientific in species_blacklist:
            continue

        txt_dct = make_txt(taxon, common) #, classes)
        img = load_img(filepath).resize(resize_size)
```

```
        # writing iNat
        sink.write({"__key__": global_id, "jpg": img, **txt_dct} #, "classes":
            classes} # REMOVED "classes", unused list of numbers, throws an error
            as an unknown extension with webdataset
                  )


#########
# BIOSCAN
#########


def copy_bioscan_from_part(sink, part):
    logger = logging.getLogger(f"p{os.getpid()}")

    db = evobio10m_reproduce.get_db(db_path)
    select_stmt = "SELECT evobio10m_id, part, filename FROM bioscan;"
    evobio10m_id_lookup = {
        (part, filename): evobio10m_id
        for evobio10m_id, part, filename in db.execute(select_stmt).fetchall()
    }
    db.close()

    name_lookup =
        naming_reproduce.load_name_lookup(disk_reproduce.bioscan_name_lookup_json)

    partdir = os.path.join(disk_reproduce.bioscan_root_dir, f"part{part}")
    for i, filename in enumerate(os.listdir(partdir)):
        if (part, filename) not in evobio10m_id_lookup:
            logger.warning(
                "EvoBio10m ID missing. [part: %d, filename: %s]", part, filename
            )
            continue

        global_id = evobio10m_id_lookup[(part, filename)]
        if global_id not in splits[args.split] or global_id in finished_ids:
            continue

        taxon, common, classes = name_lookup[global_id]

        if taxon.scientific in species_blacklist:
            continue

        txt_dct = make_txt(taxon, common) #, classes)
        filepath = os.path.join(partdir, filename)
        img = load_img(filepath).resize(resize_size)
```

```
        # writing BIOSCAN
        sink.write({"__key__": global_id, "jpg": img, **txt_dct} #, "classes":
            classes} # REMOVED "classes", unused list of numbers, throws an error
            as an unknown extension with webdataset
                  )


######
# MAIN
######


def check_status():
    finished_ids, bad_shards = set(), set()
    for root, dirs, files in os.walk(outdir):
        for file in files:
            if not file.endswith(".tar"):
                continue

            written = collections.defaultdict(set)
            with tarfile.open(os.path.join(root, file), "r|") as tar:
                try:
                    for member in tar:
                        global_id, *rest = member.name.split(".")
                        rest = ".".join(rest)
                        written[global_id].add(rest)
                except tarfile.TarError as err:
                    print(err)
                    continue

            # If you change make_txt, update these expected_texts
            expected_exts = {
                "scientific_name.txt",
                "taxonomic_name.txt",
                "common_name.txt",
                "sci.txt",
                "com.txt",
                "taxon.txt",
                "taxonTag.txt",
                "sci_com.txt",
                "taxon_com.txt",
                "taxonTag_com.txt",
                "jpg",
            }
            for global_id, exts in written.items():
                if exts != expected_exts:
```

```
                    # Delete all the files with this global_id. But this is
                        impossible
                    # with the .tar format. So instead, we delete this entire
                        shard.
                    bad_shards.add(os.path.join(root, file))
                    break
            else:
                # If we didn't early break, then this file is clean.
                finished_ids.update(set(written.keys()))

    return finished_ids, bad_shards


def make_txt(taxon, common):
    assert taxon is not None
    assert not taxon.empty, f"{common} has no taxon!"

    # test replacing the two if statements with
    common = naming_reproduce.get_common(taxon, common)
    '''if not common:
        common = taxon.scientific
    if not common:
        common = taxon.taxonomic'''

    # ex: kingdom Animalia phylum Arthropoda class ...
    tagged = " ".join(f"{tier} {value}" for tier, value in taxon.tagged)

    # IF YOU UPDATE THE KEYS HERE, BE SURE TO UPDATE check_status() TO LOOK FOR
        ALL OF
    # THESE KEYS. IF YOU DO NOT, THEN check_status() MIGHT ASSUME AN EXAMPLE IS
        WRITTEN
    # EVEN IF NOT ALL KEYS ARE PRESENT.
    return {
        # Names
        "scientific_name.txt": taxon.scientific,
        "taxonomic_name.txt": taxon.taxonomic,
        "common_name.txt": common,
        # "A photo of"... captions
        "sci.txt": f"a photo of {taxon.scientific}.",
        "com.txt": f"a photo of {common}.",
        "taxon.txt": f"a photo of {taxon.taxonomic}.",
        "taxonTag.txt": f"a photo of {tagged}.",
        "sci_com.txt": f"a photo of {taxon.scientific} with common name
            {common}.",
        "taxon_com.txt": f"a photo of {taxon.taxonomic} with common name
            {common}.",
```

```
        "taxonTag_com.txt": f"a photo of {tagged} with common name {common}.",
    }


sentinel = "STOP"


def worker(input):
    logger = logging.getLogger(f"p{os.getpid()}")
    with wds.ShardWriter(outdir, shard_counter, digits=6, maxsize=3e9) as sink:
        for func, args in iter(input.get, sentinel):
            logger.info(f"Started {func.__name__}({', '.join(map(str, args))})")
            func(sink, *args)
            logger.info(f"Finished {func.__name__}({', '.join(map(str, args))})")


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--width", type=int, default=224, help="Width of resized images."
    )
    parser.add_argument(
        "--height", type=int, default=224, help="Height of resized images."
    )
    parser.add_argument(
        "--split", choices=["train", "val", "train_small"], default="val"
    )
    parser.add_argument("--tag", default="dev", help="The suffix for the
        directory.")
    parser.add_argument(
        "--workers", type=int, default=32, help="Number of processes to use."
    )
    args = parser.parse_args()

    # Set up some global variables that depend on CLI args.
    resize_size = (args.width, args.height)
    outdir =
        f"{evobio10m_reproduce.get_outdir(args.tag)}/{args.width}x{args.height}/{args.split}"
    os.makedirs(outdir, exist_ok=True)
    print(f"Writing images to {outdir}.")

    # db_path = f"{evobio10m_reproduce.get_outdir(args.tag)}/mapping.sqlite"
    db_path = os.path.abspath(disk_reproduce.db)

    # Load train/val/train_small splits
    splits = evobio10m_reproduce.load_splits(db_path)
```

```
# Load images already written to tar files to avoid duplicate work.
# Delete any unfinished shards.
finished_ids, bad_shards = check_status()
rootlogger.info("Found %d finished examples.", len(finished_ids))
rootlogger.warning("Found %d bad shards.", len(bad_shards))
if bad_shards:
    for shard in bad_shards:
        os.remove(shard)
        rootlogger.warning("Deleted shard %d", shard)

# Load image and species blacklists for rare species
image_blacklist, species_blacklist = load_blacklists()

# Creates a shared integer
shard_counter = multiprocessing.Value("I", 0, lock=True)

# All jobs read from this queue
task_queue = multiprocessing.Queue()

# Submit all tasks
# EOL
for imgset_name in sorted(os.listdir(disk_reproduce.eol_root_dir)):
    assert imgset_name.endswith(".tar.gz")
    imgset_path = os.path.join(disk_reproduce.eol_root_dir, imgset_name)
    task_queue.put((copy_eol_from_tar, (imgset_path,)))

# Bioscan
# 113 parts in bioscan
for i in range(1, 114):
    task_queue.put((copy_bioscan_from_part, (i,)))

# iNat
for clsdir in os.listdir(disk_reproduce.inat21_root_dir):
    task_queue.put((copy_inat21_from_clsdir, (clsdir,)))

processes = []
# Start worker processes
for i in range(args.workers):
    p = multiprocessing.Process(target=worker, args=(task_queue,))
    processes.append(p)
    p.start()

# Stop worker processes
for i in range(args.workers):
    task_queue.put(sentinel)
```

```
    for p in processes:
        p.join()
```

Which was run by the SLURM script:

```
#!/bin/bash
#SBATCH --time=08:00:00
#SBATCH --job-name=make-dataset-wds_test
#SBATCH --output=make-dataset-wds_test-%j.out
#SBATCH --error=make-dataset-wds_test-%j.err
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=32
#SBATCH --partition=hpg-dev

echo $SLURM_JOB_NAME

module load conda

conda activate bioclip-train

python scripts/evobio10m/make_wds_reproduce.py --tag CVPR-2024 --split val
    --workers $SLURM_CPUS_PER_TASK
python scripts/evobio10m/make_wds_reproduce.py --tag CVPR-2024 --split
    train_small --workers $SLURM_CPUS_PER_TASK
python scripts/evobio10m/make_wds_reproduce.py --tag CVPR-2024 --split train
    --workers $SLURM_CPUS_PER_TASK
```

Which created the wds files from all the rough .tar files containing base images. To check for bad shards, we also ran a check SLURM job :

```
#!/bin/bash
#SBATCH --job-name=check_wds
#SBATCH --output=logs/check_wds_%j.out
#SBATCH --error=logs/check_wds_%j.err
#SBATCH --time=12:00:00
#SBATCH --nodes=1

# Usage:
# sbatch --account ACCOUNT --cpus-per-task N slurm/check-wds.slurm SHARDS
# e.g.
# sbatch --account ABC123 --cpus-per-task 32 slurm/check-wds.slurm
    'data/TreeOfLife-10M/dataset/evobio10m-CVPR-2024/224x224/train/shard-{000000..000165}.tar'

module load conda
```

```
conda activate bioclip-train

shardlist=$1

if [ -z "$shardlist" ]; then
  echo "Shard list is required. Usage: sbatch --account ACCOUNT
     slurm/check_wds.slurm SHARDS"
  exit 1
fi

srun python scripts/evobio10m/check_wds.py --shardlist "$shardlist" --workers
    $SLURM_CPUS_PER_TASK > logs/bad-shards.txt
```

4 Next Week Proposal

- Finish rest of Tree-of-Life scripts on Hipergator
- Write scripts testing ToL dataset (haven't worked with webdataset files ever)
- Meet with Dr. Porto and Moritz and Thomas and plan training with ToL
- Run topk visual model zero shot on other model types
- Fill out milestone report for NFHM project for Vy