Appendix A

Catalogue collection and filtering

This appendix contains the SQL query and code required to collect and filter data that is needed to obtain the star imaging data.

A.1 CasJobs SQL query

```
SELECT specobjid, class, subclass, type, ra, dec, rerun, run, camcol, field
INTO mydb.MSS_with_duplicates
FROM SpecPhotoAll
WHERE class = 'STAR' AND NOT subclass = 'CV'
```

A.2 Removing duplicates

```
import pandas as pd

# Load the CSV file
df = pd.read_csv('MSS_with_duplicates_afarrag.csv')

# Drop duplicates and keep 1st occurrence
df_unique = df.drop_duplicates(subset=['ra', 'dec'], keep='first')

# Save resulting dataframe to CSV
df_unique.to_csv('MSS_with_duplicates_removed_afarrag.csv', index=False)

print("CSV file with unique entries created successfully.")
```

A.3 Removing entries with NULL values in rerun, run, camcol or field columns

```
import pandas as pd

data = pd.read_csv('MSS_with_duplicates_removed_afarrag.csv',
↪  delimiter=',' )

# Read ra and dec coordinates as floats
data['ra'] = data['ra'].astype(float)
data['dec'] = data['dec'].astype(float)

main_classes = ['O', 'B', 'A', 'F', 'G', 'K', 'M']
filtered_data =
↪  data[data['subclass'].str.startswith(tuple(main_classes))]
↪  .reset_index(drop=True) # Filter to only get main sequence
↪  stars
filtered_data['mainclass'] = filtered_data['subclass'].str[0]
↪  # Add main class column
filtered_data = filtered_data.dropna(subset = ['rerun', 'run',
↪  'camcol', 'field']) # Drop all entries that have null
↪  values for rerun, run, camcol and/or field, as these are
↪  required to download the fits files.
filtered_data.to_csv('MSS_with_duplicates_and_NULL_removed_afarrag.csv',
↪  index=False) # Save DF to CSV
```

A.4 Obtaining unique image entries and star count

```
import pandas as pd

data =
↪  pd.read_csv('MSS_with_duplicates_and_NULL_removed_afarrag.csv')
count_per_entry = data.groupby(['rerun', 'run', 'camcol',
↪  'field', 'mainclass']).size().reset_index(name='count')
count_per_entry_sorted =
↪  count_per_entry.sort_values(by='count', ascending=False)
count_per_entry_sorted.to_csv
↪  ('MSS_unique_cam_entry_counts_with_duplicates_and_NULL_removed_afarrag.csv',
↪  index=False)
```

A.5 Selecting image entries

A.5.1 Selecting with restriction of at least 10000 samples per class, and 3601 samples for class O

```
import pandas as pd

# Read the CSV file
df =
↪  pd.read_csv('MSS_unique_cam_entry_counts_with_duplicates_and_NULL_removed
↪  _afarrag.csv')

# Group by mainclass
grouped = df.groupby('mainclass')

# Function to select entries until sum reaches the threshold,
↪  prioritizing higher counts
# Higher counts per file means less individual fits files that
↪  have to be downloaded
def select_entries(group, threshold):
    current_sum = 0
    sorted_group = group.sort_values('count',
    ↪  ascending=False).reset_index(drop=True)
    selected_rows = []

    for _, row in sorted_group.iterrows():
        if current_sum + row['count'] <= threshold:
            selected_rows.append(row)
            current_sum += row['count']
        elif current_sum < threshold:
            # If adding the full count would exceed the
            ↪  threshold,
            # we create a new row with the remaining count
            ↪  needed
            # This results in sometimes taking a few entries
            ↪  over 10000
            remaining = threshold - current_sum
            new_row = row.copy()
            new_row['count'] = remaining
            selected_rows.append(new_row)
            current_sum = threshold

        if current_sum == threshold:
            break
```

```
    return pd.DataFrame(selected_rows)

# Process each group
result_list = []
for name, group in grouped:
    if name == '0':
        threshold = 3601
    else:
        threshold = 10000

    result_list.append(select_entries(group, threshold))

# Combine all results to a single df
result = pd.concat(result_list, ignore_index=True)

# Save the result
result.to_csv('MSS_final_filtered_camera_data4.csv',
↪  index=False)
```

A.5.2 Selecting randomly

```
import pandas as pd

count_per_entry_sorted =
↪  pd.read_csv('unique_fits_entries_with_counts_sorted.csv')
count_per_entry_sorted =
↪  count_per_entry_sorted.sample(frac=1).reset_index(drop=True)

target_count = 42200
sampled_entries = []
current_count = 0

for _, row in count_per_entry_sorted.iterrows():
    if current_count >= target_count:
        break
    entry_count = row['count']
    sampled_entries.append(row)
    current_count += entry_count

# Convert sampled entries to DataFrame
sampled_entries_df = pd.DataFrame(sampled_entries)
sampled_entries_df = sampled_entries_df[['rerun', 'run',
↪  'camcol', 'field']].astype(int)
```

```
sampled_entries_df.to_csv('sampled_frame_entries_with_63700_stars.csv',
↪  index=False)
```

A.6 Converting individual camera frame entries to their star entries

```
import pandas as pd

# Read the result CSV (the one with the entries we want to
↪  match)
df_result = pd.read_csv('MSS_final_filtered_camera_data4.csv')

# Read the larger CSV file that we want to filter
df_large =
↪  pd.read_csv('MSS_with_duplicates_and_NULL_removed_afarrag.csv')

# Create a set of unique combinations from the result CSV
unique_combinations = set(df_result[['rerun', 'run', 'camcol',
↪  'field', 'mainclass']].itertuples(index=False, name=None))

# Filter the larger DataFrame based on these combinations
filtered_df = df_large[df_large.set_index(['rerun', 'run',
↪  'camcol', 'field',
↪  'mainclass']).index.isin(unique_combinations)]

# Save the filtered result to a new CSV file
filtered_df.to_csv('MSS_final_filtered_camera_data5.csv',
↪  index=False)
```

A.7 Creating training, validation and testing sets

```
import pandas as pd
import numpy as np

# Load the data
df = pd.read_csv('MSS_final_filtered_camera_data5.csv')

# Function to split the data for each mainclass
def split_data(df, train_ratio=0.7, validation_ratio=0.1,
↪  test_ratio=0.2):
    train_list = []
    validation_list = []
```

```
    test_list = []

    for mainclass in df['mainclass'].unique():
        class_df = df[df['mainclass'] == mainclass]
        class_df = class_df.sample(frac=1,
        ↪  random_state=42).reset_index(drop=True)  # Shuffle
        ↪  the data

        train_end = int(train_ratio * len(class_df))
        validation_end = train_end + int(validation_ratio *
        ↪  len(class_df))

        train_list.append(class_df[:train_end])

        ↪  validation_list.append(class_df[train_end:validation_end])
        test_list.append(class_df[validation_end:])

    train_df = pd.concat(train_list).reset_index(drop=True)
    validation_df =
    ↪  pd.concat(validation_list).reset_index(drop=True)
    test_df = pd.concat(test_list).reset_index(drop=True)

    return train_df, validation_df, test_df

# Split the data
train_df, validation_df, test_df = split_data(df)

# Save to CSV files
train_df.to_csv('MSS_training_entries.csv', index=False)
validation_df.to_csv('MSS_validation_entries.csv',
↪  index=False)
test_df.to_csv('MSS_testing_entries.csv', index=False)
```

A.8 Getting unique camera frame entries for FITS downloading

```
import pandas as pd

# Load the CSV file into a DataFrame
df = pd.read_csv('MSS_training_entries.csv')

# Select the relevant columns and drop duplicates to get
↪  unique combinations
```

```
unique_combinations = df[['run', 'rerun', 'camcol',
↪  'field']].drop_duplicates()
unique_combinations.to_csv('MSS_training_unique_cam_entries.csv',
↪  index=False)
```

Appendix B

Imaging Data Collection

This appendix contains the code required to retrieve and generate the imaging data.

B.1 RGB image downloads

```
import pandas as pd
import requests
import os
from concurrent.futures import ThreadPoolExecutor,
↪  as_completed
import time

# Read CSV file into DataFrame
data = pd.read_csv('MSS_training_entries.csv', delimiter=',')

# Read ra and dec celestial coordinates as floats
data['ra'] = data['ra'].astype(float)
data['dec'] = data['dec'].astype(float)

os.makedirs('Training_RGB', exist_ok=True)  # Create the
↪  directory if it doesn't exist

log_file = 'Training_RGB_downloading.txt' # Log file for
↪  downloading, avoids out of memory errors in browser

# Function to download images and classify
def download_image(ra, dec, mainclass, idx):
    url =
    ↪  f"http://skyserver.sdss.org/dr18/SkyServerWS/ImgCutout/getjpeg?
    ↪  ra={ra}&dec={dec}&scale=0.025&width=224&height=224"
```

```
        class_dir = os.path.join('Training_RGB', mainclass)  #
        ↪  Create a subdirectory for each star class
        os.makedirs(class_dir, exist_ok=True)  # Create the
        ↪  subdirectory if it doesn't exist
        image_path = os.path.join(class_dir,
        ↪  f'{mainclass}_class_image_ra_{ra}_dec_{dec}_rgb_image.png')

        with open(log_file, 'a') as log:
            if not os.path.exists(image_path): # Check if the
            ↪  image already exists
                response = requests.get(url)
                with open(image_path, 'wb') as f:
                    f.write(response.content)
                log.write(f"Downloaded image {idx} to
                ↪  {image_path}\n")
            else:
                log.write(f"Image {idx} already exists at
                ↪  {image_path}, skipping download.\n")

        # Print time after every 10000 downloads
        if idx > 0 and idx % 10000 == 0:
            print(f"{time.strftime('%Y-%m-%d %H:%M:%S')} -
            ↪  Downloaded {idx} images")

    # Function to handle parallel downloading
    def parallel_download(data):
        with ThreadPoolExecutor(max_workers=16) as executor:
            futures = []
            for idx, row in data.iterrows():
                future = executor.submit(download_image,
                ↪  row['ra'], row['dec'], row['mainclass'], idx)
                futures.append(future)

            for future in as_completed(futures):
                try:
                    future.result()
                except Exception as e:
                    with open(log_file, 'a') as log:
                        log.write(f"An error occurred: {e}\n")

    print(f"Started: {time.strftime('%Y-%m-%d %H:%M:%S')}")
    parallel_download(data) # For Training
```

B.2 FITS downloading

```
import os
import pandas as pd
import concurrent.futures
from tqdm import tqdm # Loading bar

def download_file(url, save_dir):
    os.makedirs(save_dir, exist_ok=True)
    command = f"wget -nc -P {save_dir} {url}" # -nc to skip
    ↪  already downloaded files
    os.system(command)

def generate_urls_and_dirs(df):
    for i in range(len(df)):
        rerun = int(df['rerun'][i])
        run1 = int(df['run'][i])
        run2 = str(int(run1)).zfill(6)
        camcol = int(df['camcol'][i])
        field = str(int(df['field'][i])).zfill(4)

        for band in ['u', 'g', 'r', 'i', 'z']:
            image_url =
            ↪  f"https://dr18.sdss.org/sas/dr18/prior-surveys/
            ↪  sdss4-dr17-eboss/photoObj/frames/{rerun}/{run1}/
            ↪  {camcol}/frame-{band}-{run2}-{camcol}-{field}.fits.bz2"
            image_save_dir =
            ↪  f"training_fits_images_{band}_band_bz2/"
            yield (image_url, image_save_dir)

sampled_unique_fits_entries =
↪  pd.read_csv('MSS_training_unique_cam_entries.csv') # Read
↪  unique fits image values into DataFrame
total_downloads = len(sampled_unique_fits_entries) * 5  # 5
↪  image bands

# Use ThreadPoolExecutor to perform parallel downloads
with concurrent.futures.ThreadPoolExecutor(max_workers=16) as
↪  executor:
    futures = []
    for url, save_dir in
    ↪  generate_urls_and_dirs(sampled_unique_fits_entries):
        futures.append(executor.submit(download_file, url,
        ↪  save_dir))
```

```
    for _ in tqdm(concurrent.futures.as_completed(futures),
    ↪  total=total_downloads, desc="Downloading files"): #
    ↪  progress bar
        pass
```

B.3 Decompressing bz2

```
import os
import bz2
import gzip
import shutil
from concurrent.futures import ThreadPoolExecutor
from tqdm import tqdm

def unpack_file(filepath, dest_dir):
    filename = os.path.basename(filepath)
    if filepath.endswith('.bz2'):
        newpath = os.path.join(dest_dir, filename[:-4])  #
        ↪  Remove .bz2 extension
        try:
            with bz2.open(filepath, 'rb') as source,
            ↪  open(newpath, 'wb') as dest:
                shutil.copyfileobj(source, dest)
            os.remove(filepath)  # Remove the original
            ↪  compressed file
        except EOFError: # Catch the EOFError
            print(f"Corrupted file detected: {filepath}.") #
            ↪  Print corrupted filename
            os.remove(filepath) # Remove the corrupted file
    elif filepath.endswith('.gz'): # For catalogue files
        newpath = os.path.join(dest_dir, filename[:-3])  #
        ↪  Remove .gz extension
        try:
            with gzip.open(filepath, 'rb') as source,
            ↪  open(newpath, 'wb') as dest:
                shutil.copyfileobj(source, dest)
            os.remove(filepath)  # Remove the original
            ↪  compressed file
        except EOFError: # Catch the EOFError
            print(f"Corrupted file detected: {filepath}.")
            ↪  Print corrupted filename
            os.remove(filepath) # Remove the corrupted file
```

```
    else:
        print(f"Unsupported file format: {filepath}") #
        ↪ Windows file system sometimes puts hidden files in
        ↪ the directory

# Get all files from a directory that have to be unpacked
def unpack_directory(directory, dest_dir):
    for root, _, files in os.walk(directory):
        for file in files:
            if file.endswith(('.bz2', '.gz')):
                yield os.path.join(root, file), dest_dir

# List of directories to process
directories = [
    ("validation_fits_images_u_band_bz2/",
    ↪ "validation_unpacked_fits_images_u_band/"),
    ("validation_fits_images_g_band_bz2/",
    ↪ "validation_unpacked_fits_images_g_band/"),
    ("validation_fits_images_r_band_bz2/",
    ↪ "validation_unpacked_fits_images_r_band/"),
    ("validation_fits_images_i_band_bz2/",
    ↪ "validation_unpacked_fits_images_i_band/"),
    ("validation_fits_images_z_band_bz2/",
    ↪ "validation_unpacked_fits_images_z_band/")
]

files_to_unpack = []
for src_dir, dest_dir in directories:
    os.makedirs(dest_dir, exist_ok=True)  # Create destination
    ↪ directory if it doesn't exist
    files_to_unpack.extend(unpack_directory(src_dir,
    ↪ dest_dir))

# Use ThreadPoolExecutor for parallel processing
with ThreadPoolExecutor(max_workers=16) as executor:
    list(tqdm(executor.map(lambda x: unpack_file(*x),
    ↪ files_to_unpack), total=len(files_to_unpack),
    ↪ desc="Unpacking files")) # *x unpacks the tuple into 2
    ↪ arguments
print("All files have been unpacked.")
```

B.4 Cropping stars out of full frame FITS images

```
import pandas as pd
from astropy.io import fits
from astropy.wcs import WCS
from astropy.nddata import Cutout2D
from astropy.coordinates import SkyCoord
import astropy.units as u
import os
import glob
from astropy.time import Time
from datetime import datetime
import warnings
from astropy.utils.exceptions import AstropyWarning

# There are deprecated columns in the FITS file headers.
↪  They're fixed automatically but print warnings, so I
↪  disabled the printing
warnings.filterwarnings('ignore', category=AstropyWarning,
↪  append=True)

os.makedirs("training_cropped_images 64x64", exist_ok=True)

df = pd.read_csv('MSS_training_entries.csv')
df['run'] = df['run'].astype(int).astype(str).apply(lambda x:
↪  x.zfill(6)) # Conversion required for filename
print(df)
df['field'] = df['field'].astype(int).astype(str).apply(lambda
↪  x: x.zfill(4)) # Conversion required for filename
df['camcol'] = df['camcol'].astype(int)
bands = ['u', 'g', 'r', 'i', 'z']

# Iterate through each star in the CSV
for _, star in df.iterrows():
    for band in bands:
        # Construct the filename pattern
        fits_file =
        ↪  f"training_unpacked_fits_images_{band}_band/
        ↪  frame-{band}-{star['run']}-{star['camcol']}-{star['field']}.fits"

        if os.path.exists(fits_file):
            try:
                with fits.open(fits_file) as hdul:
                    image_data = hdul[0].data
```

```
                    header = hdul[0].header
                    wcs = WCS(header) # Get WCS information
                    ↪ from the FITS header
                    ↪ https://docs.astropy.org/en/stable/wcs/
                    star_coord =
                    ↪ SkyCoord(ra=star['ra']*u.degree,
                    ↪ dec=star['dec']*u.degree) # SkyCoord
                    ↪ object for the star
                    cutout = Cutout2D(image_data, star_coord,
                    ↪ (64, 64), wcs=wcs) # Make the crop

                    # Create a new FITS Header Data Unit with
                    ↪ the cropped data
                    new_hdu = fits.PrimaryHDU(cutout.data)

                    ↪ new_hdu.header.update(cutout.wcs.to_header())
                    new_filename = f"training_cropped_images
                    ↪ 64x64/{star['mainclass']}_class_cutout
                    ↪ _specobjid_{star['specobjid']}_
                    ↪ ra_{star['ra']}_dec_{star['dec']}_band_{band}.fits"
                    new_hdu.writeto(new_filename,
                    ↪ overwrite=True) # Save the new FITS
                    ↪ file
                except Exception as e:
                    print(f"Error processing {fits_file}: {e}")
                    continue
            else:
                print(fits_file)
                print(f"No matching file found for star
                ↪ {star['specobjid']}, ra {star['ra']}, dec
                ↪ {star['dec']}, run {star['run']}, field
                ↪ {star['field']}, camcol {star['camcol']} in
                ↪ band {band}")
```

B.5 Synthesizing RGB images from individual band images

```
from tqdm import tqdm
import os
import numpy as np
from astropy.visualization import make_lupton_rgb
from astropy.io import fits
import matplotlib.pyplot as plt
```

```
from skimage.transform import resize
import gc
from concurrent.futures import ProcessPoolExecutor,
↪  as_completed

def normalize_image(image):
    scale_min, scale_max = np.percentile(image, [1, 99])
    scaled = np.clip((image - scale_min) / (scale_max -
    ↪  scale_min), 0, 1)
    return scaled

def combine_to_rgb(g_or_u_image, r_image, i_or_z_image, Q,
↪  alpha):
    # We resize to avoid images on edges of the full frames
    ↪  being cropped into smaller than 64x64 images
    g_or_u = normalize_image(resize(g_or_u_image, (64, 64),
    ↪  preserve_range=True, anti_aliasing=True))
    r = normalize_image(resize(r_image, (64, 64),
    ↪  preserve_range=True, anti_aliasing=True))
    i_or_z = normalize_image(resize(i_or_z_image, (64, 64),
    ↪  preserve_range=True, anti_aliasing=True))
    return make_lupton_rgb(g_or_u, r, i_or_z, Q=Q,
    ↪  stretch=alpha) # Expects float, see
    ↪  https://docs.astropy.org/en/stable/visualization/
    ↪  rgb.html#astropy-visualization-rgb



def process_fits_file(filename):
    with fits.open(filename, memmap=True) as hdul: # allows
    ↪  the array data of each HDU to be accessed with mmap,
    ↪  rather than being read into memory all at once
    ↪  https://docs.astropy.org/en/stable/io/fits/
        return hdul[0].data.copy()

def process_image(prefix, input_directory, output_directory,
↪  band_set, Q, alpha):
    bands = {}
    for band in band_set:
        filename = os.path.join(input_directory,
        ↪  f"{prefix}_band_{band}.fits")
        if os.path.exists(filename):
            bands[band] = process_fits_file(filename)
```

```
    if all(band in bands for band in band_set):
        band1 = bands[band_set[0]]
        band2 = bands[band_set[1]]
        band3 = bands[band_set[2]]
        rgb_image = combine_to_rgb(band1, band2, band3, Q,
        ↪  alpha)
        plt.imsave(os.path.join(output_directory,
        ↪  f'{prefix}_{band_set}_RGB.png'), rgb_image,
        ↪  format='png', dpi=300)
        return True
    return False

# Process the images in batches to avoid crashing due to
↪  insufficient memory
def process_batch(batch, Q, alpha):
    results = []
    for item in batch:
        prefix, input_directory, gri_output, urz_output, Q,
        ↪  alpha = item
        gri_result = process_image(prefix, input_directory,
        ↪  gri_output, 'gri', Q, alpha)
        urz_result = process_image(prefix, input_directory,
        ↪  urz_output, 'urz', Q, alpha)
        results.append((gri_result, urz_result))
    return results

def process_directory(input_directory, gri_output_directory,
↪  urz_output_directory, Q, alpha):
    os.makedirs(gri_output_directory, exist_ok=True)
    os.makedirs(urz_output_directory, exist_ok=True)

    prefixes = set()
    for filename in os.listdir(input_directory):
        if filename.endswith('.fits'):
            prefix = filename.rsplit('_band_', 1)[0] #
            ↪  maxsplit parameter to 1, will return a list
            ↪  with 2 elements
            ↪  https://www.w3schools.com/python/ref_string_rsplit.asp
            prefixes.add(prefix)

    batch_size = 50
    batches = [list(prefixes)[i:i+batch_size] for i in
    ↪  range(0, len(prefixes), batch_size)]
```

```
    with ProcessPoolExecutor(max_workers=16) as executor:
        futures = []
        for batch in batches:
            batch_items = [(prefix, input_directory,
            ↪  gri_output_directory, urz_output_directory, Q,
            ↪  alpha) for prefix in batch]
            futures.append(executor.submit(process_batch,
            ↪  batch_items, Q, alpha))

        with tqdm(total=len(prefixes) * 2, desc="Processing
        ↪  images", unit="step") as pbar:
            for future in as_completed(futures):
                results = future.result()
                for gri_result, urz_result in results:
                    if gri_result:
                        pbar.update(1)
                    if urz_result:
                        pbar.update(1)

process_directory('training_cropped_images 64x64',
↪  'training_gri_q_8_a_0.02 64x64', 'training_urz_q_8_a_0.02
↪  64x64', 8, 0.02)
process_directory('validation_cropped_images 64x64',
↪  'validation_gri_q_8_a_0.02 64x64',
↪  'validation_urz_q_8_a_0.02 64x64', 8, 0.02)
process_directory('testing_cropped_images 64x64',
↪  'testing_gri_q_8_a_0.02 64x64', 'testing_urz_q_8_a_0.02
↪  64x64', 8, 0.02)
```

B.6 Sorting images into mainclass folders

```
import os
import shutil

def sort_files_by_class(directory):
    for filename in os.listdir(directory):
        file_path = os.path.join(directory, filename)

        if not os.path.isfile(file_path):
            continue

        star_class = filename[0].upper()
        subdirectory_path = os.path.join(directory,
        ↪  star_class)
```

```
        os.makedirs(subdirectory_path, exist_ok=True)
        shutil.move(file_path, os.path.join(subdirectory_path,
        ↪  filename))



sort_files_by_class("training_gri_q_8_a_0.02 64x64")
sort_files_by_class("training_urz_q_8_a_0.02 64x64")
```

B.7 Side by side GRI-URZ images

```
import os
from PIL import Image

# Define directories
gri_dir = 'training_gri_q_8_a_0.02 64x64'
urz_dir = 'training_urz_q_8_a_0.02 64x64'
output_dir = 'training_side_by_side_q_8_a_0.02'
os.makedirs(output_dir, exist_ok=True)

def get_image_pairs(gri_dir, urz_dir, class_label):
    gri_class_dir = os.path.join(gri_dir, class_label)
    urz_class_dir = os.path.join(urz_dir, class_label)

    gri_images = [f for f in os.listdir(gri_class_dir) if
    ↪  f.endswith('.png')]
    urz_images = [f for f in os.listdir(urz_class_dir) if
    ↪  f.endswith('.png')]
    image_pairs = list(zip(gri_images, urz_images))  # Create
    ↪  image pairs

    return image_pairs, gri_class_dir, urz_class_dir

for class_label in ['O', 'B', 'A', 'F', 'G', 'K', 'M']:
    output_class_dir = os.path.join(output_dir, class_label)
    os.makedirs(output_class_dir, exist_ok=True)
    image_pairs, gri_class_dir, urz_class_dir =
    ↪  get_image_pairs(gri_dir, urz_dir, class_label)

    for gri_image_name, urz_image_name in image_pairs:
        gri_image_path = os.path.join(gri_class_dir,
        ↪  gri_image_name)
```

```
        urz_image_path = os.path.join(urz_class_dir,
        ↪  urz_image_name)
        gri_image = Image.open(gri_image_path)
        urz_image = Image.open(urz_image_path)

        # Combine the images side by side
        combined_img = Image.new('RGB', (128, 64))
        combined_img.paste(gri_image, (0, 0))
        combined_img.paste(urz_image, (64, 0))

        base_name =
        ↪  os.path.basename(gri_image_path).replace('_gri_RGB.png',
        ↪  '_side_by_side.png')
        output_path = os.path.join(output_class_dir,
        ↪  base_name)
        combined_img.save(output_path)

        print(f'Saved {output_path}')
```

B.8 Upscaling images

```
from PIL import Image
import os
import time


# https://pillow.readthedocs.io/en/stable/
↪  handbook/concepts.html#filters-comparison-table
def upscale_and_save_image(image_path, output_path):
    try:
        if os.path.exists(output_path): # Skip if image exists
            return

        with Image.open(image_path) as image:
            upscaled_image = image.resize((224, 224),
            ↪  Image.LANCZOS) # Resize image to 224x224
            upscaled_image.save(output_path)

    except Exception as e:
        print(f"Error processing {image_path}: {e}")

input_dir = 'training_gri_q_8_a_0.02 64x64'
```

```
output_dir = 'training_gri_q_8_a_0.02 224x224 from 64x64'

counter = 0
for root, dirs, files in os.walk(input_dir):
    for file in files:
        input_path = os.path.join(root, file)
        os.makedirs(output_dir, exist_ok=True)
        output_path = os.path.join(output_dir, file) # output
        ↪ path

        try:
            upscale_and_save_image(input_path, output_path)
            counter += 1
            if counter % 10000 == 0:
                print(f"{time.strftime('%Y-%m-%d %H:%M:%S')} -
                ↪ Upscaled {counter} images")

        except Exception as e:
            print(f"Error processing {input_path}: {e}")

print(f"All images have been upscaled and saved to
↪ {output_dir}")
```

Appendix C

ViT

This appendix contains the code to set up the ViT.

C.1 Preparing and saving DatasetDict object for ViT

```
from datasets import load_dataset, DatasetDict

# Load the training, validation and testing sets
train_dataset = load_dataset("imagefolder",
↪  data_dir="Training_RGB")
validation_dataset = load_dataset("imagefolder",
↪  data_dir="Validation_RGB")
test_dataset = load_dataset("imagefolder",
↪  data_dir="Testing_RGB")

# Create a DatasetDict directly with the train, validation and
↪  testing datasets.
dataset_dict = DatasetDict({
    'train': train_dataset['train'],
    'validation': validation_dataset['train'],
    'test': test_dataset['train']
})

dataset_dict.save_to_disk('dataset_dictionary_RGB')
```

C.2 Loading DatasetDict object

```
dataset_dict = load_from_disk("dataset_dictionary_RGB")
```

C.3 Importing ViT

```
from transformers import ViTImageProcessor
model_name_or_path = 'google/vit-base-patch16-224-in21k'
processor =
↪  ViTImageProcessor.from_pretrained(model_name_or_path)
```

C.4 Transforming images to ViT inputs

```
def transform(example_batch):
    # Take a list of PIL images and turn them to pixel values
    rgb_images = [img for img in example_batch['image']] # Use
    ↪  this for native RGB images
    # rgb_images = [img.convert('RGB') for img in
    ↪  example_batch['image']] # Use this for Synthesized RGB
    ↪  images
    inputs = processor(rgb_images, return_tensors='pt')

    # Don't forget to include the labels!
    inputs['labels'] = example_batch['label']
    return inputs

prepared_ds = dataset_dict.with_transform(transform)
```

C.5 Data Collator

```
import torch

def collate_fn(batch):
    return {
        'pixel_values': torch.stack([x['pixel_values'] for x
        ↪  in batch]),
        'labels': torch.tensor([x['labels'] for x in batch])
    }
```

C.6 Evaluation Metrics

```
import numpy as np
from datasets import load_metric

accuracy_metric = load_metric("accuracy")
precision_metric = load_metric("precision")
recall_metric = load_metric("recall")
```

```
f1_metric = load_metric("f1")


def compute_metrics(p):
    preds = np.argmax(p.predictions, axis=1) # Get the
    ↪  predicted class
    labels = p.label_ids # Labels

    # Calculate accuracy, precision, recall, f1 (with
    ↪  average='weighted' for multi-class and to account for
    ↪  label imbalance)
    accuracy = accuracy_metric.compute(predictions=preds,
    ↪  references=labels)
    precision = precision_metric.compute(predictions=preds,
    ↪  references=labels, average='weighted')
    recall = recall_metric.compute(predictions=preds,
    ↪  references=labels, average='weighted')
    f1 = f1_metric.compute(predictions=preds,
    ↪  references=labels, average='weighted')

    # Return dictionary of metrics
    return {
        "accuracy": accuracy["accuracy"],
        "precision": precision["precision"],
        "recall": recall["recall"],
        "f1": f1["f1"]
    }
```

C.7 Classification Head

```
from transformers import ViTForImageClassification

labels = dataset_dict['train'].features['label'].names

model = ViTForImageClassification.from_pretrained(
    model_name_or_path,
    num_labels=len(labels),
    id2label={str(i): c for i, c in enumerate(labels)},
    label2id={c: str(i) for i, c in enumerate(labels)}
)
```

C.8 Training arguments

```
from transformers import TrainingArguments

training_args = TrainingArguments(
  output_dir="./vit-stellar-classification-RGB",
  per_device_train_batch_size=32,
  eval_strategy="steps",
  num_train_epochs=50,
  fp16=True,
  save_steps=1000,
  eval_steps=1000,
  logging_steps=1000,
  learning_rate=1e-5,
  save_total_limit=10,
  metric_for_best_model="accuracy",
  remove_unused_columns=False,
  push_to_hub=False,
  report_to='tensorboard',
  load_best_model_at_end=True,
)
```

C.9 Early stopping

```
from transformers import EarlyStoppingCallback

early_stopping_callback = EarlyStoppingCallback(
    early_stopping_patience=5,
    early_stopping_threshold=0.001
)
```

C.10 Trainer

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=collate_fn,
    compute_metrics=compute_metrics,
    train_dataset=prepared_ds["train"],
    eval_dataset=prepared_ds["validation"],
    tokenizer=processor,
```

```
    callbacks=[early_stopping_callback]
)
```

C.11 Start training

```
train_results = trainer.train()
trainer.save_model()
trainer.log_metrics("train", train_results.metrics)
trainer.save_metrics("train", train_results.metrics)
trainer.save_state()
```

C.12 Evaluate model using test set

```
test_results = trainer.predict(prepared_ds["test"])
trainer.log_metrics("test", test_results.metrics)
trainer.save_metrics("test", test_results.metrics)
```

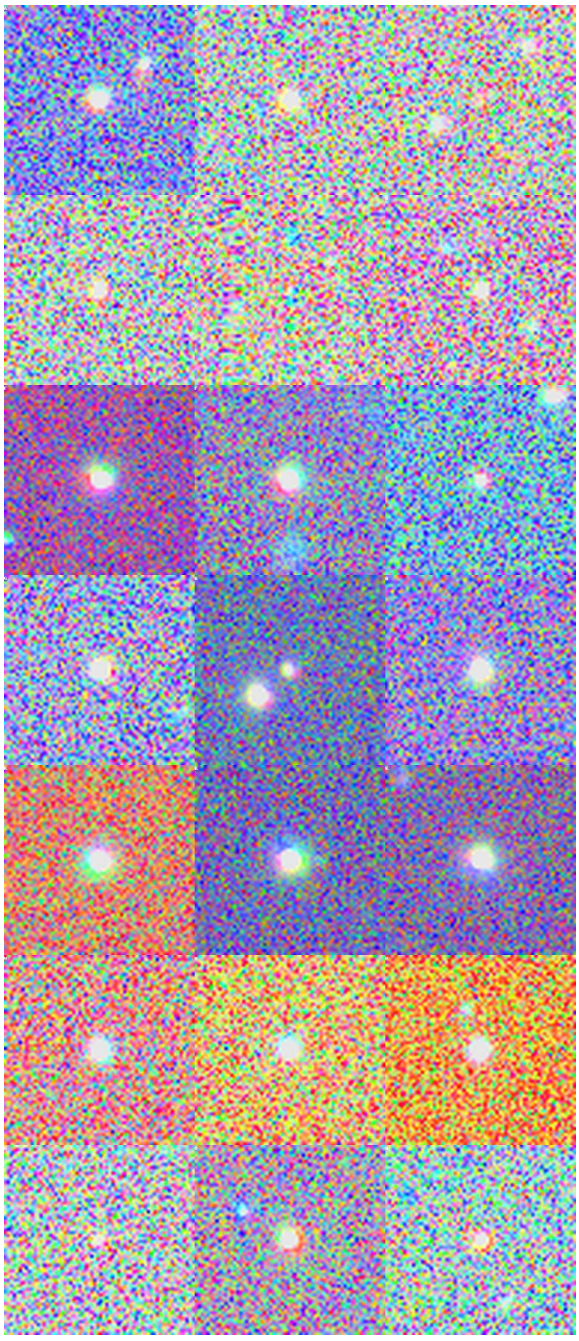
Appendix D

Images

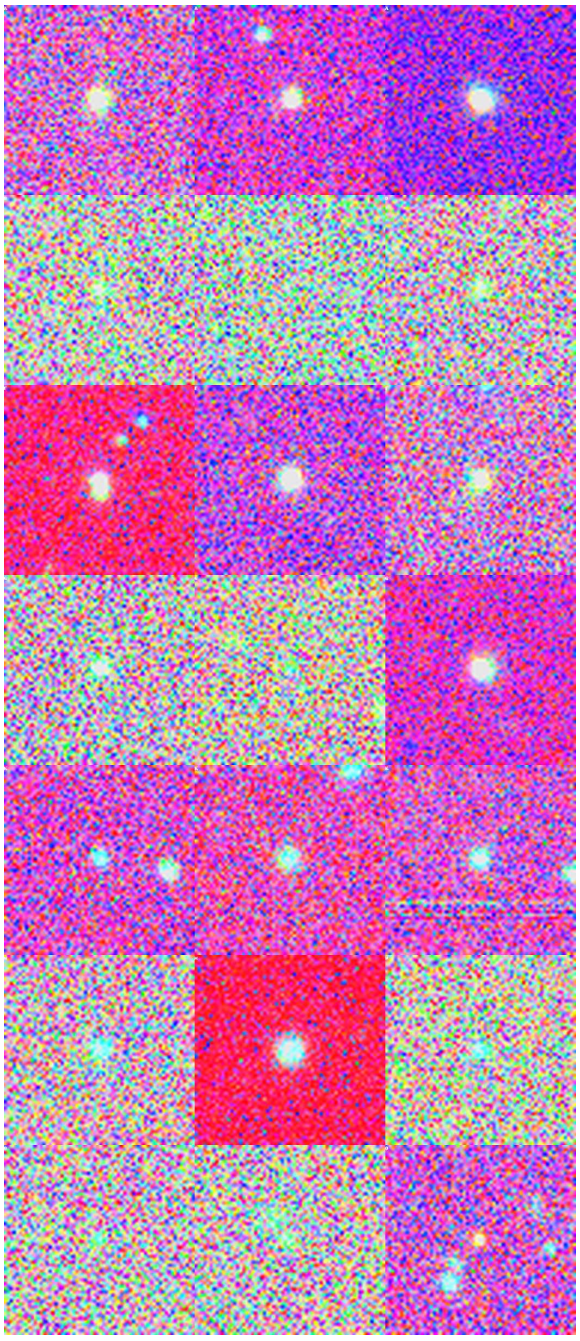
This appendix contains example images that are used. In each section there are 3 images per row. Each row is a star class. The stars appear in the class order A, B, F, G, K, M, O

D.1 RGB images

D.2 GRI images



D.3 URZ images



D.4 GRI-URZ side-by-side images