Template Based Matching

ICS3206 - Course Project (Jupyter Notebook #2)

Name: Andrea Filiberto Lucas
ID No: 0279704L

Importing Necessary Libraries

Essential libraries such as OpenCV, NumPy, Matplotlib, IPyWidgets, and Pillow are imported in this section. The script checks for these packages and installs any that are missing using `pip`, ensuring the environment is properly configured for the subsequent code cells.

```
############################
#          IMPORTS         #
############################

import importlib
import subprocess
import sys

# ANSI escape codes for colored text
GREEN = "\033[92m"
RED = "\033[91m"
RESET = "\033[0m"

# List of packages to check and their corresponding import names
packages = {
    'opencv-python': 'cv2',
    'numpy': 'numpy',
    'matplotlib': 'matplotlib.pyplot',
    'seaborn': 'seaborn',
    'pandas': 'pandas',
    'ipywidgets': 'ipywidgets',
    'Pillow': 'PIL',
}

def install_package(package_name):
    """
    Install a package using pip.
    """
    try:
        print(f"Installing package: {package_name}")
        subprocess.check_call([sys.executable, "-m", "pip", "install",
```

```
package_name])
        print(f"{GREEN}Successfully installed {package_name}.{RESET}")
    except subprocess.CalledProcessError as e:
        # Print the error message in red
        print(f"{RED}Failed to install package {package_name}. Error:
{e}{RESET}")
        sys.exit(1)

def check_and_install_packages(packages_dict):
    """
    Check if the packages are installed, and install them if they are
missing.
    """
    for package, import_name in packages_dict.items():
        try:
            # Attempt to import the package
            importlib.import_module(import_name.split('.')[0])
            print(f"{RESET}Package '{package}' is already installed.
{RESET}")
        except ImportError:
            print(f"{RED}Package '{package}' is not installed.
{RESET}")
            install_package(package)

# Check and install packages
check_and_install_packages(packages)

# Now, import the packages after ensuring they are installed
try:
    import cv2
    import numpy as np
    import matplotlib.pyplot as plt
    import seaborn as sns
    import pandas as pd
    from concurrent.futures import ThreadPoolExecutor, as_completed
    from sklearn.metrics import confusion_matrix, accuracy_score
    import threading
    import sys
    import os
    import glob
    import ipywidgets as widgets
    from IPython.display import display, clear_output
    from PIL import Image
    from sklearn.metrics import confusion_matrix,
ConfusionMatrixDisplay
except ImportError as e:
    print(f"{RED}An error occurred while importing packages: {e}
{RESET}")
    sys.exit(1)
```

```python
# Print the success message in green
print(f"{GREEN}All required packages are installed and imported successfully.{RESET}")
```

```
Package 'opencv-python' is already installed.
Package 'numpy' is already installed.
Package 'matplotlib' is already installed.
Package 'seaborn' is already installed.
Package 'pandas' is already installed.
Package 'ipywidgets' is already installed.
Package 'Pillow' is already installed.
All required packages are installed and imported successfully.
```

Settings

This section defines the configuration parameters and settings essential for this jupyter notebook. It includes a function to select sky images from a specified directory using an interactive dropdown widget. Additionally, various thresholds and constraints are set for the template matching process, such as the ratio test threshold, minimum number of good matches, scaling factors for templates, and bounding box area limits.

```python
#############################
#        SETTINGS           #
#############################

def select_sky_image(temp_dir="../Temp/"):
    """
    Display a dropdown widget to select an image from the specified directory.
    """
    import os
    import glob
    import sys
    from IPython.display import display, clear_output
    import ipywidgets as widgets

    # Supported image extensions
    image_extensions = ['*.png', '*.jpg', '*.jpeg', '*.bmp']

    # List to hold all found image paths
    image_files = []

    # Search for images with supported extensions
    for ext in image_extensions:
        image_files.extend(glob.glob(os.path.join(temp_dir, ext)))

    # Filter out non-file paths (just in case)
    image_files = [f for f in image_files if os.path.isfile(f)]
```

```
    # If no images are found, display an error message
    if not image_files:
        print(f"No images found in the directory '{temp_dir}'. Please add images and try again.")
        sys.exit(1)

    # Sort the images alphabetically
    image_files.sort()

    # Create a list of image filenames for display
    image_names = [os.path.basename(f) for f in image_files]

    # Create a Dropdown widget with increased width and adjusted description width
    dropdown = widgets.Dropdown(
        options=image_names,
        description='Select Image:',
        disabled=False,
        layout=widgets.Layout(width='350px'),  # Increased width for the dropdown
        style={'description_width': '85px'}   # Adjusted description width
    )

    # Create an Output widget to display the selected image path
    out = widgets.Output()

    # Define the callback function when a selection is made
    def on_change(change):
        """
        Display the selected image path when a new image is selected.
        """
        if change['type'] == 'change' and change['name'] == 'value':
            with out:
                clear_output()
                selected_image = os.path.join(temp_dir, change['new'])
                print(f"Selected image: {selected_image}")
                # Update the global variable
                global sky_image_path
                sky_image_path = selected_image

    # Attach the callback to the Dropdown
    dropdown.observe(on_change)

    # Display the widgets
    display(dropdown, out)

    # Initialize the sky_image_path with the first image and display it in the output widget
```

```
    if image_files:
        sky_image_path = image_files[0]
        with out:
            print(f"Default selected image: {sky_image_path}")

    return sky_image_path

# Set the sky_image_path dynamically based on user selection
sky_image_path = select_sky_image("../Temp/")  # Updated path for image selection

template_dir = "../8CD/"  # Folder containing sub-folders with constellation templates

# Ratio test threshold for Lowe's test (stricter => 0.6 to 0.7)
RATIO_TEST_THRESH = 0.7

# Minimum number of good matches
MIN_GOOD_MATCHES = 15

# Scales to try for each template
SCALES = [0.6, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4]

# Bounding box area constraints (to discard wildly incorrect homographies)
MIN_BBOX_AREA = 100   # e.g., discard if warped area < 100 pixels
MAX_BBOX_AREA = 1e7   # e.g., discard if warped area > 10 million pixels
```


{"model_id":"ae8edc1721194e858a004cba37dd830c","version_major":2,"version_minor":0}

{"model_id":"88830ea3c14b494fb57d37bbc95e1159","version_major":2,"version_minor":0}

Global Progress

This section manages the tracking of the overall progress during template processing. The total number of templates is calculated by scanning all relevant directories, and a shared counter is initialized to monitor the number of processed templates. A thread-safe mechanism is implemented to update and display a dynamic progress bar, providing real-time feedback on the percentage of completion. This ensures efficient monitoring of the processing workflow and helps maintain transparency throughout the project's execution.

What is Shoelace Formula?

The Shoelace Formula is a mathematical method for calculating the area of a polygon by using the coordinates of its vertices. It involves multiplying and summing the coordinates in a specific pattern that resembles shoelace lacing.

```python
############################
#     GLOBAL PROGRESS       #
############################

# Calculate the total number of templates (files) across all
subfolders
all_template_files = []
for folder in os.listdir(template_dir):
    folder_path = os.path.join(template_dir, folder)
    if os.path.isdir(folder_path):
        for f in os.listdir(folder_path):
            if not f.startswith("."):  # ignore hidden files
                full_path = os.path.join(folder_path, f)
                all_template_files.append((folder, full_path))

total_templates = len(all_template_files)
progress_lock = threading.Lock()
templates_processed = 0  # shared counter

def update_progress():
    """
    Safely update the global progress and print a single progress bar
    showing how many templates have been processed out of the total.
    """
    with progress_lock:
        global templates_processed
        templates_processed += 1
        percentage = (templates_processed / total_templates) * 100
        bar = create_progress_bar(percentage)
        sys.stdout.write(f"\rProcessing Templates: {percentage:5.1f}%
{bar} ({templates_processed}/{total_templates})")
        sys.stdout.flush()

############################
#       HELPER METHODS      #
############################

def load_image(image_path):
    """Load an image from the given path and convert it to
grayscale."""
    if not os.path.exists(image_path):
        print(f"[ERROR] File does not exist: {image_path}")
        return None
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        print(f"[ERROR] Could not load image at {image_path}")
    return image

def preprocess_image(image):
    """Apply histogram equalization if contrast is low."""
```

```
    if image is None:
        return None
    if cv2.meanStdDev(image)[1] < 50:
        return cv2.equalizeHist(image)
    return image

def display_image(image, title="Image"):
    """Display an image (grayscale) via matplotlib."""
    plt.figure(figsize=(10, 8))
    plt.imshow(image, cmap='gray')
    plt.title(title)
    plt.axis('off')
    plt.show()

def create_progress_bar(percentage, width=50):
    """Generate a simple text-based progress bar."""
    filled = int(width * percentage / 100) if percentage else 0
    return "█" * filled + "░" * (width - filled)

def polygon_area(corners):
    """
    Compute the polygon area using the Shoelace formula.
    corners should be a list or array of shape (4, 2).
    """
    x = corners[:, 0]
    y = corners[:, 1]
    # Shoelace formula
    return 0.5 * np.abs(np.dot(x, np.roll(y, 1)) - np.dot(y,
np.roll(x, 1)))
```

Main Matching Logic

This section implements the core functionality for matching constellation templates with the sky image (previously selected image). The sky image is loaded and preprocessed, and the ORB feature detector is initialized to identify keypoints and descriptors. For each template, multiple scales are applied to account for size variations, and keypoints are matched using a FLANN-based matcher with Lowe's ratio test to ensure robust correspondences. Homography is computed using RANSAC to filter out outliers, and bounding box area constraints are enforced to validate the matches. A scoring mechanism evaluates the quality of each match, and progress is updated in real-time to monitor the matching process efficiently.

What is ORB Features?

ORB (Oriented FAST and Rotated BRIEF) is a feature detection and description algorithm used to identify and describe keypoints in images. It is efficient and robust, making it suitable for real-time applications. ORB combines the FAST keypoint detector and the BRIEF descriptor with additional modifications to handle rotation and improve performance.

What is a FLANN-based Matcher with Lowe's Ratio?

FLANN (Fast Library for Approximate Nearest Neighbors) is an optimized library for fast nearest neighbor searches in large datasets. When combined with Lowe's ratio test, it efficiently matches feature descriptors by finding the two nearest neighbors for each descriptor and accepting the match only if the distance of the closest neighbor is significantly lower than the second closest. This helps in reducing false matches and improving matching accuracy.

What is Homography and RANSAC?

Homography is a transformation matrix that maps points from one plane to another, allowing for the alignment of images taken from different perspectives. RANSAC (Random Sample Consensus) is an iterative algorithm used to estimate the homography by identifying inliers among the matched keypoints, thereby filtering out outliers. Together, they enable the accurate alignment and matching of features between images despite noise and mismatches.

```
#############################
#     MAIN MATCHING LOGIC   #
#############################

# Load the sky image once
sky_image = load_image(sky_image_path)
if sky_image is None:
    print("[FATAL] Sky image not found or could not be loaded.")
    sys.exit(1)

sky_image = preprocess_image(sky_image)
sky_h, sky_w = sky_image.shape

# Initialize ORB
orb = cv2.ORB_create(nfeatures=2000)

def match_keypoints(template_img, sky_img):
    """
    1. Detect and compute ORB features for both images.
    2. Use FLANN-based matcher and Lowe's ratio test.
    3. Use RANSAC to filter out outliers and compute homography.
    4. Discard matches if bounding box is out of realistic area range.
    Returns:
        kp1, kp2, good_matches, mean_dist, homography
    """
    kp1, des1 = orb.detectAndCompute(template_img, None)
    kp2, des2 = orb.detectAndCompute(sky_img, None)

    if des1 is None or des2 is None:
        return kp1, kp2, [], None, None

    # Convert to float32 if needed
    if des1.dtype != np.float32:
        des1 = des1.astype(np.float32)
```

```
    if des2.dtype != np.float32:
        des2 = des2.astype(np.float32)

    # FLANN for ORB (LSH Index)
    index_params = dict(algorithm=1, trees=10)
    search_params = dict(checks=50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)

    # KNN match
    matches = flann.knnMatch(des1, des2, k=2)

    # Lowe's ratio test
    good_matches = []
    for m, n in matches:
        if m.distance < RATIO_TEST_THRESH * n.distance:
            good_matches.append(m)

    # If enough matches, try homography
    M = None
    mean_dist = None
    if len(good_matches) >= MIN_GOOD_MATCHES:
        src_pts = np.float32([kp1[m.queryIdx].pt for m in
good_matches]).reshape(-1, 1, 2)
        dst_pts = np.float32([kp2[m.trainIdx].pt for m in
good_matches]).reshape(-1, 1, 2)

        M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,
5.0)
        if M is not None and mask is not None:
            # Keep only inliers
            mask = mask.ravel().astype(bool)
            inlier_matches = [good_matches[i] for i, val in
enumerate(mask) if val]
            good_matches = inlier_matches

            # Compute average distance of inlier matches
            distances = [m.distance for m in good_matches]
            mean_dist = float(np.mean(distances)) if distances else
None

            # Optional bounding-box check
            h_t, w_t = template_img.shape
            corners = np.float32([[0, 0],
                                  [w_t, 0],
                                  [w_t, h_t],
                                  [0,   h_t]]).reshape(-1, 1, 2)

            # Warp the template's corners onto the sky
            warped_corners = cv2.perspectiveTransform(corners, M)
            warped_corners = warped_corners.reshape(-1, 2)
```

```
            # Calculate area
            area = polygon_area(warped_corners)
            # If area is out of plausible range, discard these matches
            if not (MIN_BBOX_AREA <= area <= MAX_BBOX_AREA):
                good_matches = []
                M = None

    return kp1, kp2, good_matches, mean_dist, M

def resize_template(template, scale_factor):
    """Resize template by the given scale factor."""
    new_w = int(template.shape[1] * scale_factor)
    new_h = int(template.shape[0] * scale_factor)
    return cv2.resize(template, (new_w, new_h))

def compute_match_score(num_good_matches, mean_dist):
    """
    Example scoring function:
      score = num_good_matches / (mean_dist + 1)
    The lower the mean distance, the higher the score.
    The more good matches, the higher the score.
    """
    if mean_dist is None:  # If no distance is computed, score = 0
        return 0.0
    return float(num_good_matches) / (mean_dist + 1.0)

def process_single_template(folder_name, template_path):
    """
    Load a single template, preprocess it, try multiple scales,
    pick the best scale's match among them, and return result.
    """
    best_result = None  # Stores: (score, num_good_matches, scale,
template_img, kp1, kp2, matches, matched_vis)
    template_img = load_image(template_path)
    if template_img is None:
        return None
    template_img = preprocess_image(template_img)

    # Iterate over multiple scales
    for scale in SCALES:
        resized = resize_template(template_img, scale)
        kp1, kp2, g_matches, mean_dist, M = match_keypoints(resized,
sky_image)

        if g_matches:
            num_good = len(g_matches)
            current_score = compute_match_score(num_good, mean_dist)
            if (best_result is None) or (current_score >
best_result[0]):
```

```
                # Draw a quick visualization for potential final use
                matched_vis = None
                if kp1 and kp2 and g_matches:
                    matched_vis = cv2.drawMatches(
                        resized, kp1, sky_image, kp2, g_matches, None,

flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
                    )
                best_result = (
                    current_score,
                    num_good,
                    scale,
                    folder_name,
                    os.path.basename(template_path),
                    matched_vis
                )

    update_progress()
    return best_result
```

Gather & Process Data

In this section, template matching results are gathered and processed efficiently. The total number of template files is determined, and a `ThreadPoolExecutor` is utilized to concurrently process each template. Each template is analyzed individually, and the outcomes are collected into a results list. Once all templates have been processed, the results are sorted in descending order based on their matching scores, preparing the data for the selection of the top matches.

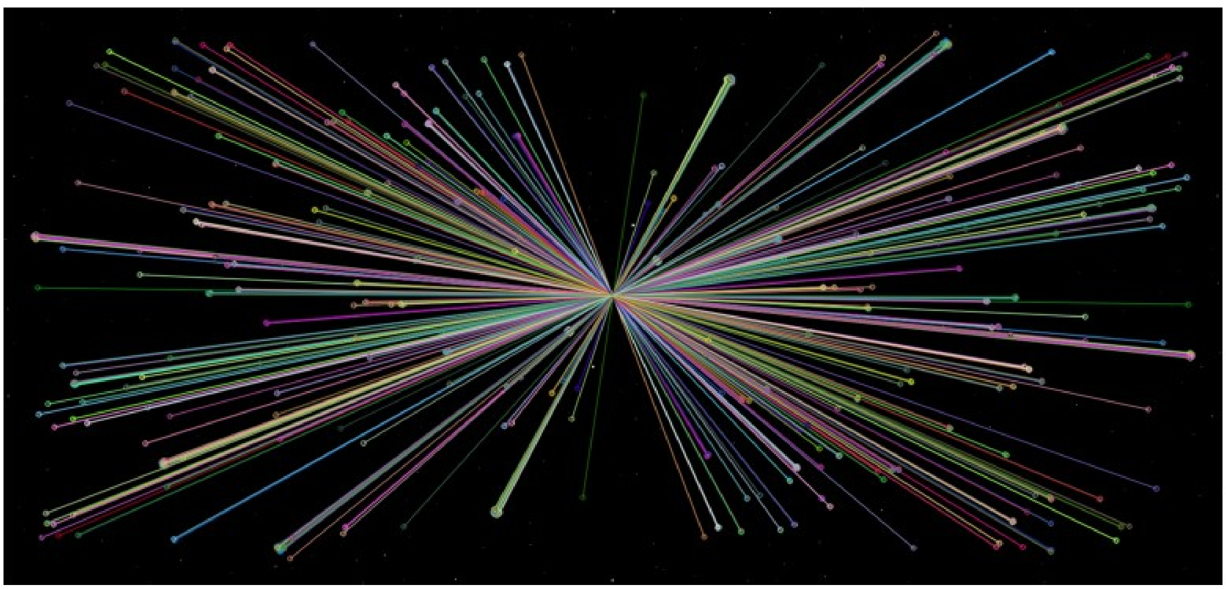
```
#############################
#   GATHER & PROCESS DATA   #
#############################

# Prepare to gather match results
results = []

def worker(args):
    """Helper function for concurrency: process a single template."""
    folder_name, template_path = args
    return process_single_template(folder_name, template_path)

print(f"Found {total_templates} total template files to process in
'{template_dir}'.")

# Use a ThreadPoolExecutor to process each template concurrently
with ThreadPoolExecutor() as executor:
    futures = [executor.submit(worker, tpl) for tpl in
all_template_files]
    for fut in as_completed(futures):
        outcome = fut.result()
```

```
        if outcome is not None:
            results.append(outcome)

# Normalize folder names (e.g., treat "CoronaBorealis" and
"CoronaBorealis_augmented" as the same)
def normalize_folder_name(folder_name):
    """
    Normalize the folder name by removing '_augmented' suffix or any
other distinction.
    """
    return folder_name.split("_")[0]  # Keep only the base name before
any underscores

# Sort by the final computed score in descending order
# Each result is (score, num_good, scale, folder_name,
template_filename, matched_vis)
results.sort(key=lambda x: x[0], reverse=True)

#############################
#   SELECT & PRINT TOP 3    #
#############################

# Pick the top result from each constellation folder (3 unique
constellations)
seen_folders = set()
top_3 = []

for res in results:
    score, num_good, scale, folder, tpl_file, matched_vis = res
    normalized_folder = normalize_folder_name(folder)
    if normalized_folder not in seen_folders:
        seen_folders.add(normalized_folder)
        top_3.append(res)
        if len(top_3) == 3:
            break

print("\n\n\033[1mFinished Processing\033[0m\n")

# Display the top 3 matches
for rank, (score, num_good, scale, folder, tpl_file, matched_vis) in
enumerate(top_3, start=1):
    probability = (num_good / float(sky_h * sky_w)) * 100

    print(f"Rank {rank}: Constellation:
{normalize_folder_name(folder)} | Template: {tpl_file}")
    print(f"  Scale: {scale} | Good Matches: {num_good} | Score:
{score:.2f}")
    print(f"  Approx Probability: {probability:.4f}%\n")

    if matched_vis is not None:
```

```
        display_image(matched_vis, title=f"Rank {rank} - {normalize_folder_name(folder)}")
```

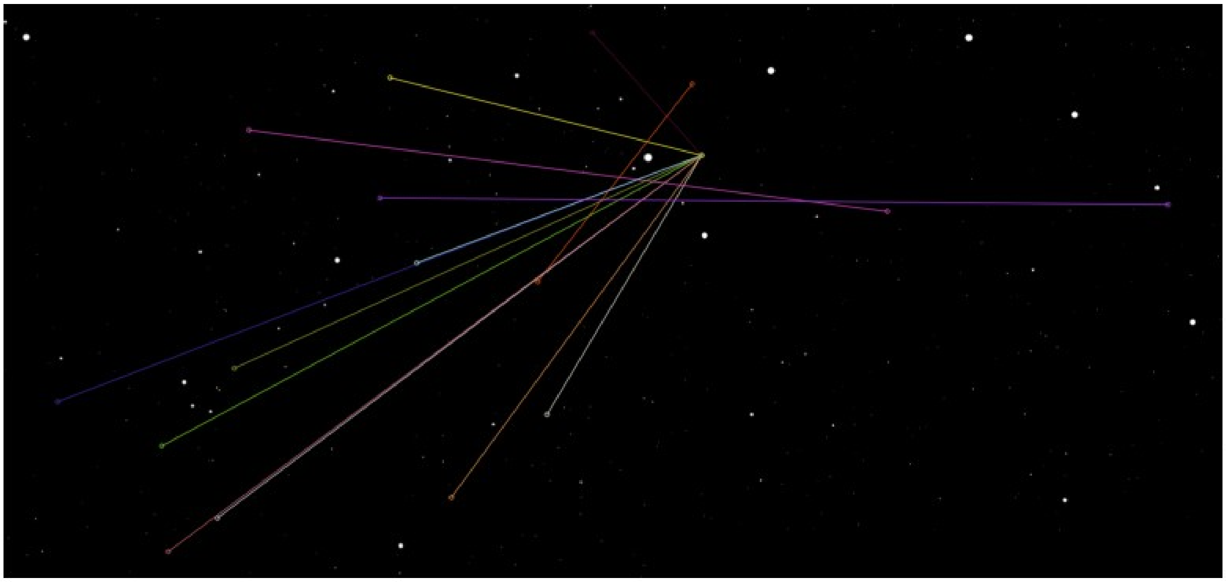
```
Found 1952 total template files to process in '../8CD/'.
Processing Templates: 100.0%
██████████████████████████████████████████████████ (1952/1952)

Finished Processing

Rank 1: Constellation: CoronaBorealis | Template: CoronaBorealis-AbsoluteClean.png
  Scale: 1.0 | Good Matches: 451 | Score: 451.00
  Approx Probability: 0.0779%
```

Rank 1 - CoronaBorealis



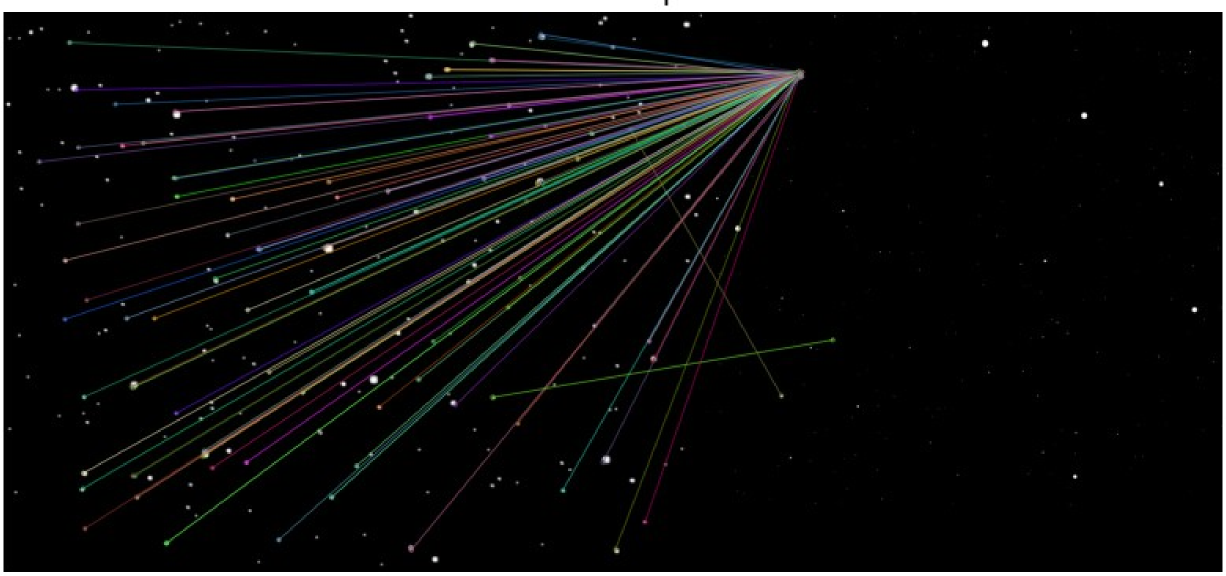
```
Rank 2: Constellation: LeoMinor | Template: LeoMinor-LightSky.png
  Scale: 1.1 | Good Matches: 13 | Score: 9.61
  Approx Probability: 0.0022%
```

Rank 2 - LeoMinor



```
Rank 3: Constellation: Lepus | Template: Lepus-SKYMAP_blur_ksize_3.png
  Scale: 1.3 | Good Matches: 96 | Score: 3.13
  Approx Probability: 0.0166%
```

Rank 3 - Lepus



Results Summary & Visualization

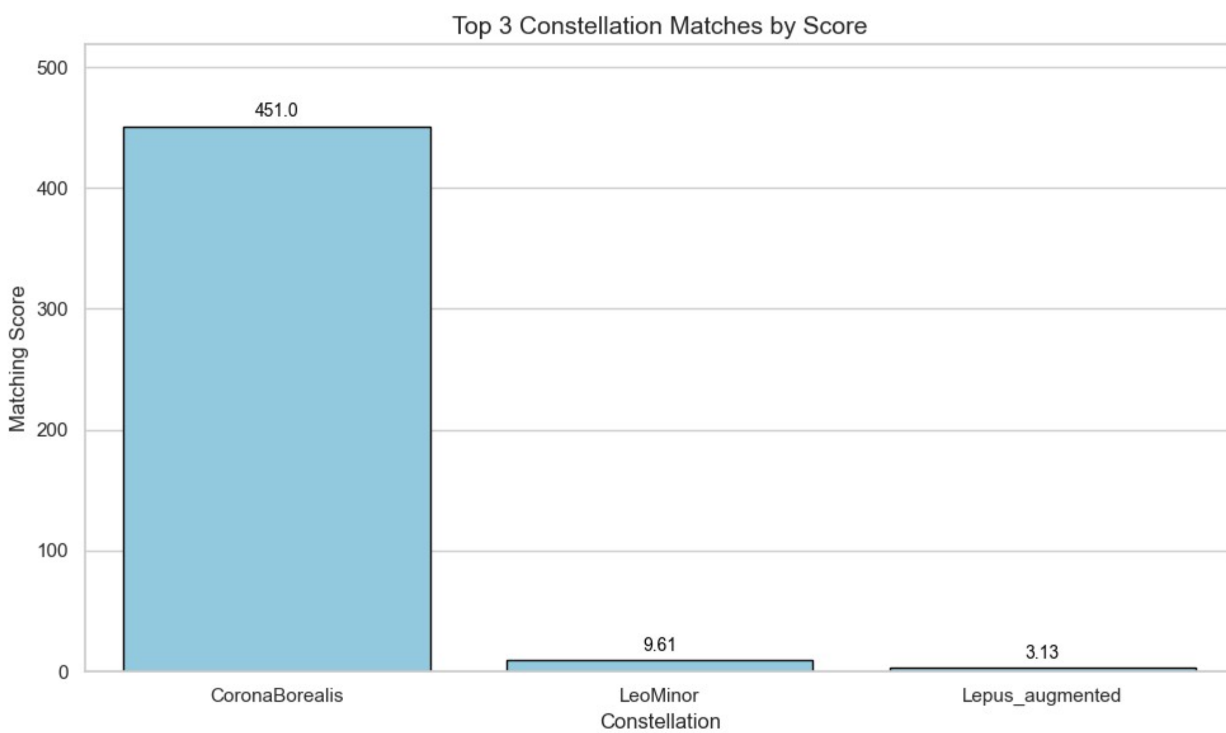
This section compiles and visualizes the results of the template matching process. A summary table of the top three constellation matches is created and displayed using a pandas DataFrame. Additionally, bar charts are generated with Seaborn to illustrate the matching scores of the top matches and the average scores per constellation. These visualizations provide a clear and comprehensive overview of the matching performance, facilitating easy comparison and analysis of the results.

```
#####################################
#   RESULTS SUMMARY & VISUALIZATION  #
#####################################

summary_data = [
    {
        'Rank': rank,
        'Constellation': folder,
        'Template File': tpl_file,
        'Scale Factor': scale,
        'Good Matches': num_good,
        'Score': round(score, 2),
        'Approx Probability (%)': round((num_good / (sky_h * sky_w)) * 
100, 4)
    }
    for rank, (score, num_good, scale, folder, tpl_file, matched_vis) 
in enumerate(top_3, start=1)
]

# Create DataFrame
summary_df = pd.DataFrame(summary_data)

# Display the summary table
print("Top 3 Constellation Matches:")
display(summary_df)

# Visualization settings
sns.set(style="whitegrid")

# Plot Top 3 Constellation Scores
plt.figure(figsize=(10, 6))
barplot = sns.barplot(
    x='Constellation',
    y='Score',
    data=summary_df,
    color='skyblue',  # Changed from palette to color
    edgecolor='black'
)

# Add value labels on top of each bar
for index, row in summary_df.iterrows():
    barplot.text(
        index,
        row['Score'] + summary_df['Score'].max() * 0.01,
        f"{row['Score']}",
        color='black',
        ha="center",
```

```
        va='bottom',
        fontsize=10
    )

plt.xlabel('Constellation', fontsize=12)
plt.ylabel('Matching Score', fontsize=12)
plt.title('Top 3 Constellation Matches by Score', fontsize=14)
plt.ylim(0, summary_df['Score'].max() * 1.15)  # Add space above the
highest bar
plt.tight_layout()
plt.show()

# Calculate Average Scores per Constellation using pandas
results_df = pd.DataFrame(results, columns=['Score', 'Num_Good',
'Scale', 'Constellation', 'Template_File', 'Matched_Vis'])
avg_scores_df = results_df.groupby('Constellation')
['Score'].mean().reset_index()
avg_scores_df.rename(columns={'Score': 'Average Score'}, inplace=True)
avg_scores_df.sort_values(by='Average Score', ascending=False,
inplace=True)

# Display the average scores table
print("Average Matching Scores per Constellation:")
display(avg_scores_df)

# Plot Average Matching Scores per Constellation
plt.figure(figsize=(12, 8))
barplot_avg = sns.barplot(
    x='Constellation',
    y='Average Score',
    data=avg_scores_df,
    color='orange',  # Changed from palette to color
    edgecolor='black'
)

# Add value labels on top of each bar
for index, row in avg_scores_df.iterrows():
    barplot_avg.text(
        index,
        row['Average Score'] + avg_scores_df['Average Score'].max() *
0.01,
        f"{row['Average Score']:.2f}",
        color='black',
        ha="center",
        va='bottom',
        fontsize=10
    )

plt.xlabel('Constellation', fontsize=12)
plt.ylabel('Average Matching Score', fontsize=12)
```

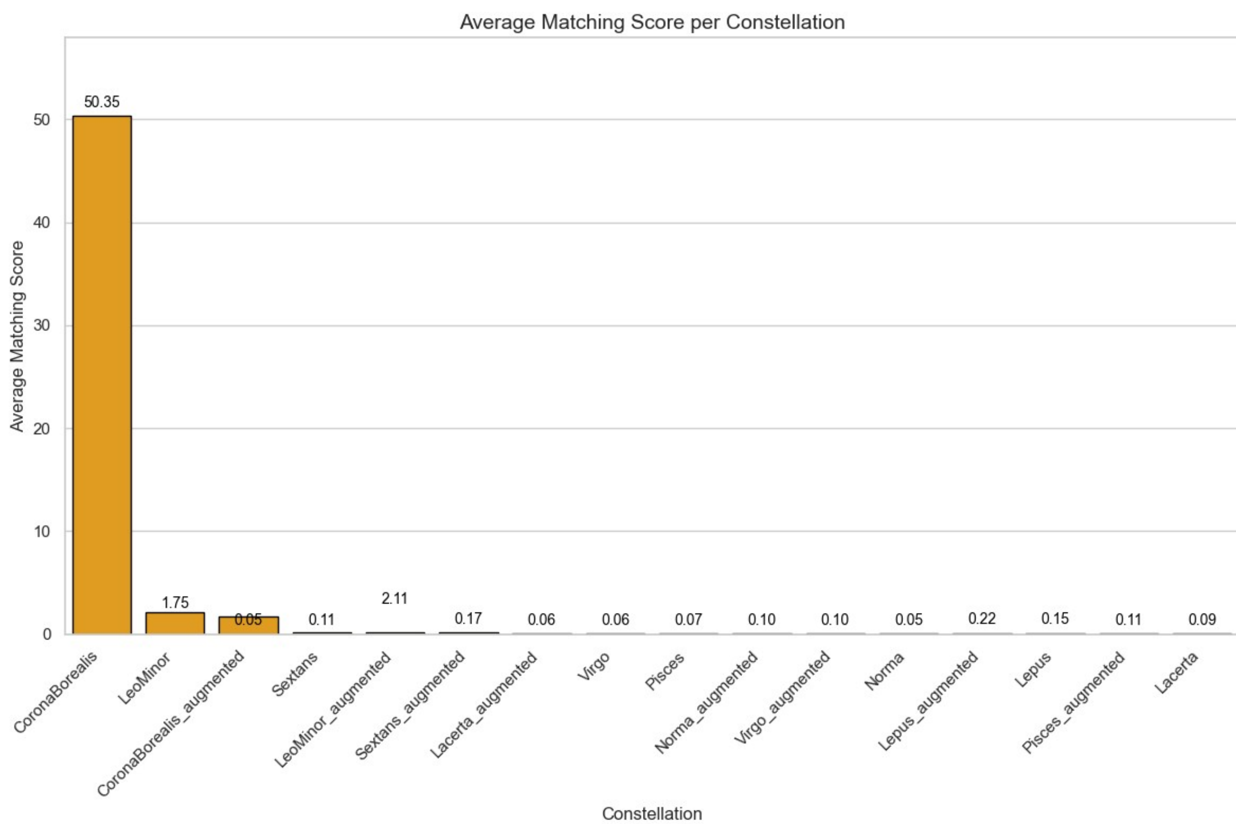
```
plt.title('Average Matching Score per Constellation', fontsize=14)
plt.xticks(rotation=45, ha='right')
plt.ylim(0, avg_scores_df['Average Score'].max() * 1.15)  # Add space
above the highest bar
plt.tight_layout()
plt.show()
```

```
Top 3 Constellation Matches:

   Rank    Constellation                     Template File  Scale
Factor  \
0     1   CoronaBorealis  CoronaBorealis-AbsoluteClean.png
1.0
1     2         LeoMinor            LeoMinor-LightSky.png
1.1
2     3  Lepus_augmented     Lepus-SKYMAP_blur_ksize_3.png
1.3

   Good Matches   Score  Approx Probability (%)
0           451  451.00                  0.0779
1            13    9.61                  0.0022
2            96    3.13                  0.0166
```




```
Average Matching Scores per Constellation:

                Constellation  Average Score
0              CoronaBorealis      50.354528
```

```
4                    LeoMinor       2.113510
1   CoronaBorealis_augmented       1.752442
12                    Sextans       0.215513
5          LeoMinor_augmented       0.165173
13          Sextans_augmented       0.153286
3           Lacerta_augmented       0.110503
14                      Virgo       0.106664
10                     Pisces       0.101527
9             Norma_augmented       0.099887
15            Virgo_augmented       0.089451
8                       Norma       0.069246
7             Lepus_augmented       0.056439
6                       Lepus       0.055386
11           Pisces_augmented       0.045424
2                     Lacerta       0.045191
```



Evaluation (Top-3 Check)

This script evaluates the accuracy of a constellation recognition system by comparing the ground truth of an image's constellation against the top-3 predicted results. It ensures that the ground truth appears in the predictions, indicating success. If the ground truth is missing from the top-3 predictions, the evaluation reports a failure. The process involves normalizing folder names from the predictions and cross-checking them with the ground truth mappings.

```python
########################################
#       EVALUATION (Top-3 Check)       #
########################################

# Dictionary mapping image filenames to the correct constellation name
ground_truth = {
    "1-CB.png":  "CoronaBorealis",
    "1-LAC.png": "Lacerta",
    "1-LM.png":  "LeoMinor",
    "1-LPS.png": "Lepus",
    "1-P.png":   "Pisces",
    "1-V.png":   "Virgo",
    "1-S.png":   "Sextans",
    "1-N.png":   "Norma",
    "2-CB.png":  "CoronaBorealis",
    "2-LAC.png": "Lacerta",
    "2-LM.png":  "LeoMinor",
    "2-LPS.png": "Lepus",
    "2-P.png":   "Pisces",
    "2-V.png":   "Virgo",
    "2-S.png":   "Sextans",
    "2-N.png":   "Norma",
}

# Get just the filename of the sky_image_path (e.g., "1-CB.png")
filename = os.path.basename(sky_image_path)
print(f"\nEvaluating the Top-3 predictions for the image: {filename}")

# Lookup the ground-truth constellation
gt_constellation = ground_truth.get(filename, None)

if gt_constellation is None:
    print(f"\nNo ground truth found for {filename}; skipping 
evaluation.\n")
else:
    # Extract predicted constellations
    predicted_constellations = []
    for result in top_3:
        if len(result) >= 4:  # Ensure we have enough elements to 
process
            folder = result[3]  # Fourth element is the folder name
            if isinstance(folder, str):
                normalized_name = normalize_folder_name(folder)
                predicted_constellations.append(normalized_name)
            else:
                print(f"Skipping invalid folder value (not a string): 
{folder}")
        else:
            print(f"Skipping invalid result due to insufficient 
elements: {result}")
```

```
    print("\n\033[1mEvaluation\033[0m")
    print(f"Ground Truth Constellation: {gt_constellation}")
    print(f"Top-3 Predicted Constellations:
{predicted_constellations}")

    # Check if the ground-truth constellation is in the Top-3
predictions
    if gt_constellation in predicted_constellations:
        print(f"\033[92mSUCCESS:\033[0m Ground-truth constellation
'{gt_constellation}' is in the Top-3 predictions.")
    else:
        print(f"\033[91mFAILURE:\033[0m Ground-truth constellation
'{gt_constellation}' did not appear in Top-3 predictions.")
```

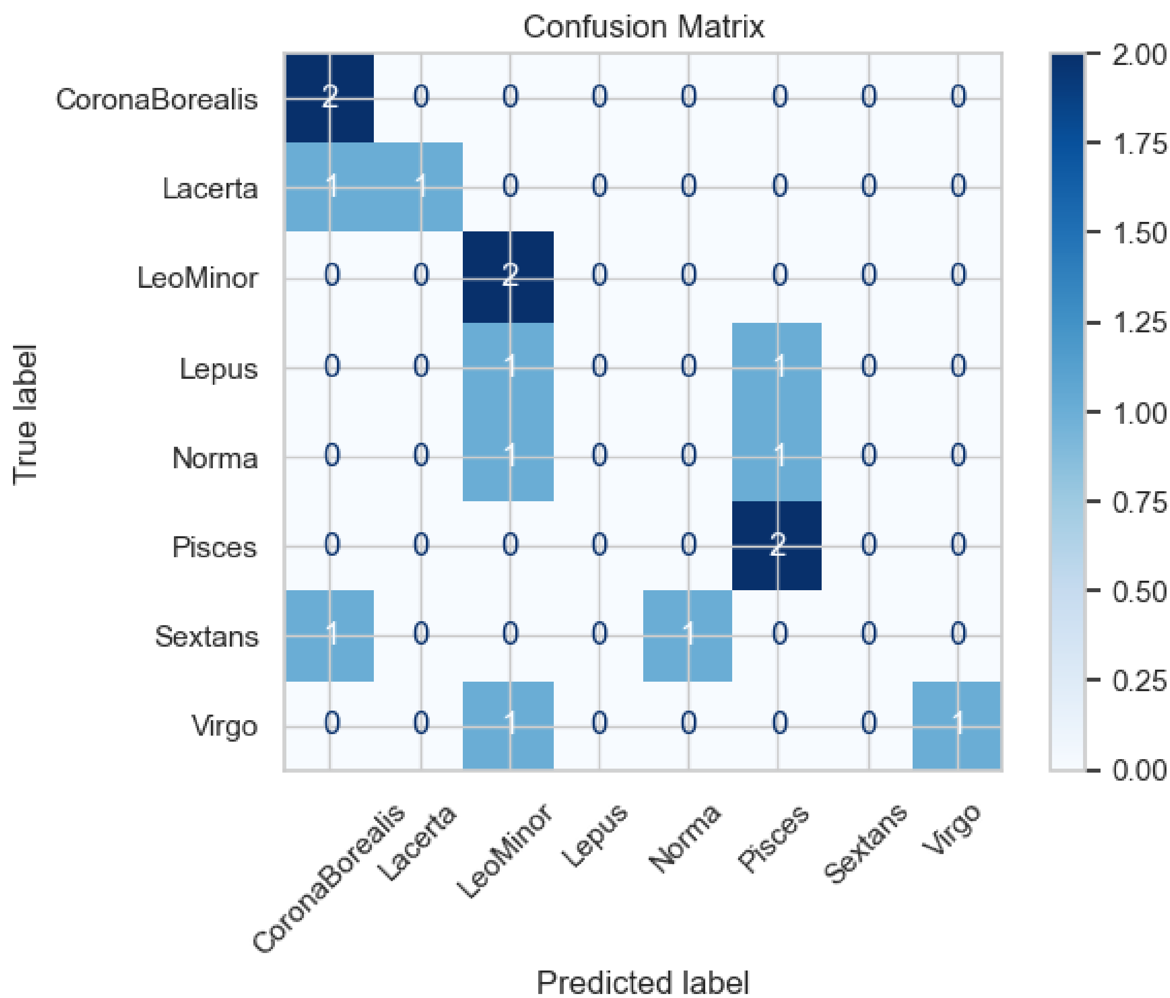
```
Evaluating the Top-3 predictions for the image: 1-CB.png

Evaluation
Ground Truth Constellation: CoronaBorealis
Top-3 Predicted Constellations: ['CoronaBorealis', 'LeoMinor',
'Lepus']
SUCCESS: Ground-truth constellation 'CoronaBorealis' is in the Top-3
predictions.
```

Confusion Matrix & Accuracy

This script evaluates the performance of a constellation recognition system by comparing true labels with manually stored predictions. The predictions were pre-computed due to kernel crashes when loops were used to process them dynamically. A confusion matrix is generated to visually analyze the classification results, and the overall accuracy is calculated to quantify the model's performance.

```
########################################
#       EVALUATION (Top-3 Check)       #
########################################

# Manually Stored Value from previous Runs (for evaluation)
'''
The predictions were done manually as when loops were implemented the
kernel was crashing due to high memory usage
'''
predictions = {
    "1-CB.png":  "CoronaBorealis",
    "1-LAC.png": "CoronaBorealis",
    "1-LM.png":  "LeoMinor",
    "1-LPS.png": "LeoMinor",
    "1-P.png":   "Pisces",
    "1-V.png":   "LeoMinor",
```

```
    "1-S.png":   "Norma",
    "1-N.png":   "Pisces",
    "2-CB.png":  "CoronaBorealis",
    "2-LAC.png": "Lacerta",
    "2-LM.png":  "LeoMinor",
    "2-LPS.png": "Pisces",
    "2-P.png":   "Pisces",
    "2-V.png":   "Virgo",
    "2-S.png":   "CoronaBorealis",
    "2-N.png":   "LeoMinor",
}

# Create lists of true and predicted labels
true_labels = [ground_truth[filename] for filename in
ground_truth.keys()]
predicted_labels = [predictions.get(filename, "Unknown") for filename
in ground_truth.keys()]

# Get the unique class labels
unique_labels = sorted(set(true_labels + predicted_labels))

# Generate the confusion matrix
cm = confusion_matrix(true_labels, predicted_labels,
labels=unique_labels)

# Display the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=unique_labels)
disp.plot(cmap=plt.cm.Blues, xticks_rotation=45)
plt.title("Confusion Matrix")
plt.show()

# Get unique labels
unique_labels = sorted(set(ground_truth.values()))

# Convert ground truth and predictions to lists
y_true = [ground_truth[filename] for filename in ground_truth.keys()]
y_pred = [predictions[filename] for filename in ground_truth.keys()]

# Calculate confusion matrix and accuracy
accuracy = accuracy_score(y_true, y_pred)

# Create a DataFrame for better visualization of the confusion matrix
cm_df = pd.DataFrame(cm, index=unique_labels, columns=unique_labels)

# Display accuracy
print(f"\nAccuracy: {accuracy * 100:.2f}%")
```




```
Accuracy: 50.00%
```