All experiments were conducted on a Windows 11 (64-bit) platform using Python 3.10. Core dependencies comprised NumPy (v1.26), pandas (v2.2), OpenCV (v4.9), scikit-image (v0.23), Matplotlib (v3.8), and Seaborn (v0.13); Excel I/O was supported by openpyxl (v3.1). Optionally, scikit-learn (v1.4) was employed to compute cosine similarity on flattened vectors. Experiments were executed on a commodity CPU without GPU acceleration. To ensure reproducibility, we fixed the random seed (SEED = 42) and standardized all images to grayscale at 256 × 256 pixels (uint8, data range 0–255) prior to similarity computation. Software and parameter configurations were held constant throughout, and package versions were recorded for traceability.

```
"""
SSIM-based facial similarity pipeline (Appendix-ready, with statistics & robustness)

Replace the following placeholders with your local paths before running:
  <AFU_IMAGE_DIR>      -> directory containing A-fu images
  <SAMPLE_IMAGE_DIR>   -> directory containing sampled comparison images
  <SAMPLE_EXCEL_PATH>  -> Excel with columns: period, religious types, subject, identifier

"""

from pathlib import Path
import random
from collections import defaultdict

import numpy as np
import pandas as pd
import cv2
from skimage.metrics import structural_similarity as ssim
import matplotlib.pyplot as plt
import seaborn as sns

from scipy.stats import mannwhitneyu

# ----------------- Global parameters -----------------
K = 3  # max samples per stratum
SEED = 42  # fixed random seed for reproducibility
AGG_METHOD = "mean"  # 'mean' or 'max' similarity across A-fu images (per sample)
N_BOOT = 1000  # bootstrap resamples for CI
RNG = np.random.default_rng(SEED)

# ----------------- Placeholders for paths -----------------
afu_image_dir = Path("<AFU_IMAGE_DIR>")  # e.g., /project/data/afutu
sample_image_dir = Path("<SAMPLE_IMAGE_DIR>")  # e.g.,
```

```
/project/data/sample_pic

sample_excel_path = Path("<SAMPLE_EXCEL_PATH>")  # e.g.,
/project/meta/sample1.xlsx

out_dir = Path("./_outputs")

out_dir.mkdir(parents=True, exist_ok=True)


random.seed(SEED)


# ----------------- 1) Read Excel -----------------

records = []

df = pd.read_excel(sample_excel_path)

for _, row in df.iterrows():

    records.append({

        'period': str(row['Period']).strip() if pd.notna(row['Period']) else
'',

        'religious types': str(row['Religious types']).strip() if
pd.notna(row['Religious types']) else '',

        'subject': str(row['Subject']).strip() if pd.notna(row['Subject'])
else '',

        'identifier': str(row['Identifier']).strip() if
pd.notna(row['Identifier']) else ''

    })


# ----------------- 2) Build stratification dict -----------------

strata = defaultdict(list)

for rec in records:

    key = '|'.join([rec['period'], rec['religious types'], rec['subject']])

    strata[key].append(rec)


# ----------------- 3) Stratified sampling -----------------

sampled = []

for layer in strata.values():

    if len(layer) <= K:

        sampled.extend(layer)

    else:

        random.shuffle(layer)  # shuffled with fixed seed for
reproducibility
```

```
        sampled.extend(layer[:K])



def preprocess_image(img_path: Path) -> np.ndarray:
    """
    Preprocess an image: read, convert to grayscale, and resize to 256x256.
    Returns uint8 array (256,256).
    """
    img = cv2.imread(str(img_path))
    if img is None:
        raise FileNotFoundError(f"Could not read image: {img_path}")
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    resized = cv2.resize(gray, (256, 256), interpolation=cv2.INTER_AREA)
    return resized



# ----------------- 4) Load A-fu images -----------------
afu_image_names, afu_image_list = [], []
for p in sorted(afu_image_dir.iterdir()):
    if p.is_file():
        image = preprocess_image(p)
        afu_image_list.append(image)
        afu_image_names.append(p.name)


# ----------------- 5) Load sampled images -----------------
sample_image_list, sample_image_names = [], []
sample_meta = []  # rows: dict(period, religious types, subject,
identifier)
for rec in sampled:
    filename = rec['identifier'].strip()
    img_path = sample_image_dir / f"{filename}.jpg"
    image = preprocess_image(img_path)
    sample_image_list.append(image)
    sample_image_names.append(filename)
    sample_meta.append(rec)
```

```
# ----------------- 6) Core: SSIM matrix -----------------
def compute_ssim_matrix(afu_images, sample_images, data_range=255,
**ssim_kwargs) -> np.ndarray:
    """
    Compute SSIM matrix (n_afus x n_samples).
    """
    n = len(afu_images)
    m = len(sample_images)
    sim_matrix = np.zeros((n, m), dtype=float)
    for i in range(n):
        for j in range(m):
            sim_matrix[i, j] = ssim(afu_images[i], sample_images[j],
data_range=data_range, **ssim_kwargs)
    return sim_matrix



# Baseline SSIM (no gaussian weights; keep same as your original to avoid
breaking changes)
matrix = compute_ssim_matrix(afu_image_list, sample_image_list)
pd.DataFrame(matrix, index=afu_image_names,
columns=sample_image_names).to_csv(out_dir / "ssim_matrix.csv")



# ----------------- 7) Aggregate per-sample similarity -----------------
def aggregate_per_sample(sim_mat: np.ndarray, method: str = "mean"):
    """
    Reduce n_afus x n_samples -> per-sample scalar via mean or max across
A-fu rows.
    Returns: np.ndarray of shape (n_samples,)
    """
    # compute mean
    if method == "mean":
        return sim_mat.mean(axis=0)
    # compute maximum
    elif method == "max":
```

```
        return sim_mat.max(axis=0)
    else:
        raise ValueError("AGG_METHOD must be 'mean' or 'max'.")



per_sample_val = aggregate_per_sample(matrix, AGG_METHOD)


# Build tidy dataframe for stats
meta_df = pd.DataFrame(sample_meta)  # period, religious types, subject, identifier
meta_df["ssim_value"] = per_sample_val
meta_df["agg_method"] = AGG_METHOD
meta_df.to_csv(out_dir / "per_sample_stats.csv", index=False, encoding='gbk')



# ----------------- 8) Statistics: Bootstrap CI, Mann-Whitney, Hedges' g ----------------
def bootstrap_ci(x: np.ndarray, n: int = N_BOOT, alpha: float = 0.05) -> tuple[float, float, float]:
    # Draw bootstrap samples with replacement from the original data and compute the mean
    boots = [RNG.choice(x, size=len(x), replace=True).mean() for _ in range(n)]
    # Compute the (1 - alpha) confidence interval
    lo, hi = np.percentile(boots, [100 * alpha / 2, 100 * (1 - alpha / 2)])
    return float(np.mean(x)), float(lo), float(hi)



def hedges_g(a: np.ndarray, b: np.ndarray) -> float:
    na, nb = len(a), len(b)
    # Check whether sample sizes are sufficient
    if na < 2 or nb < 2:
        print(f"Warning: sample size too small to compute Hedges' g (n1={na}, n2={nb})")
        return float('nan')  # or return another default value
    sa2 = a.var(ddof=1)
    sb2 = b.var(ddof=1)
```

```
    sp = np.sqrt(((na - 1) * sa2 + (nb - 1) * sb2) / (na + nb - 2))
    g = (b.mean() - a.mean()) / sp if sp > 0 else 0.0
    # small sample correction (Hedges' g)
    J = 1 - 3 / (4 * (na + nb) - 9) if (na + nb) > 2 else 1.0
    return float(g * J)



religious_types = [
    "buddhism",      # "buddhism"
    "daoism",        # "daoism"
    "comprehensive"  # "comprehensive"
]
periods = [
    "yuan dynasty",          # "yuan dynasty"
    "ming dynasty",          # "ming dynasty"
    "qing dynasty",          # "qing dynasty"
    "republic of china era"   # "republic of china era"
]


# Create grouped data
grouped_data = {}


for religious_type in religious_types:
    for period in periods:
        data = meta_df.loc[
            (meta_df["period"].str.lower() == period.lower()) &
            (meta_df["religious types"].str.lower() ==
religious_type.lower()),
            "ssim_value"
        ].to_numpy()


        if len(data) > 0:
            group_key = f"{religious_type}_{period}"
            grouped_data[group_key] = data
```

```
def compare_periods(period1_name, period1_data, period2_name,
period2_data):
    """Compare data between two periods"""
    if len(data1) + len(data2) < 2:
        print(f"Warning: {key1} and {key2} have too few observations to
compute effect size")
        return None
    if len(period1_data) > 0 and len(period2_data) > 0:
        mean1, lo1, hi1 = bootstrap_ci(period1_data)
        mean2, lo2, hi2 = bootstrap_ci(period2_data)
        stat, pval = mannwhitneyu(period1_data, period2_data,
alternative="two-sided")
        g = hedges_g(period1_data, period2_data)


        print(f"\n[{period1_name}]  mean SSIM = {mean1:.3f} (95% CI:
{lo1:.3f}-{hi1:.3f})")
        print(f"[{period2_name}]  mean SSIM = {mean2:.3f} (95% CI:
{lo2:.3f}-{hi2:.3f})")
        print(f"Mann–Whitney U p = {pval:.4g}   Hedges' g = {g:.2f}")
        return True
    else:
        print(f"Insufficient data to compare {period1_name} and
{period2_name}")
        return False



# Pairwise comparisons
group_keys = list(grouped_data.keys())
for i in range(len(group_keys)):
    for j in range(i + 1, len(group_keys)):
        key1, key2 = group_keys[i], group_keys[j]
        data1, data2 = grouped_data[key1], grouped_data[key2]
        compare_periods(key1, data1, key2, data2)



# ----------------- 9) Visualize the similarity matrix -----------------
```

```
plt.figure(figsize=(40, 30))
sns.heatmap(
    matrix,
    xticklabels=sample_image_names,
    yticklabels=afu_image_names,
    annot=True,
    cmap='coolwarm'
)
plt.title("Facial Similarity Matrix (SSIM)")
plt.tight_layout()
plt.show()


# ----------------- 10) Notes for manuscript -----------------
print("\nMANUSCRIPT NOTES:")
print("- SSIM in [0,1]; higher values indicate greater structural similarity.")
print("- We report mean SSIM with 95% bootstrap CIs (n=1000), Mann–Whitney U tests, and Hedges' g.")
```