## STEP 1: MOUNT THE DRIVE

```
from google.colab import drive
drive.mount('/content/drive')
```

## STEP 2: SETTING THE ENVIRONMENT

```
!pip install geopandas shapely pyproj
```

## STEP 3: CONVERT LOCALITY TEXT" RAS EL-HEKMA" TO COORDINATES USING POINT-RADIUS GEOREFERENCING METHOD BASED ON DARWEN CORE GUIDLEINES

```
import geopandas as gpd
from shapely.geometry import Point, Polygon


def dms_to_dd(degrees, minutes, seconds, direction):
    """Converts Degrees, Minutes, Seconds to Decimal Degrees."""
    dd = float(degrees) + float(minutes) / 60 + float(seconds) / 3600
    if direction in ["S", "W"]:
        dd *= -1
    return dd


# 1. Convert DMS to Decimal Degrees
lat_max = dms_to_dd(31, 21, 33.98, "N")  # 31.359439
lat_min = dms_to_dd(31, 2, 46.32, "N")  # 31.046200
lon_min = dms_to_dd(27, 41, 22.92, "E")  # 27.689700
lon_max = dms_to_dd(27, 59, 0.96, "E")  # 27.983600

# 2. Build Bounding Box Polygon
bbox_coords = [
    (lon_min, lat_min),
    (lon_max, lat_min),
    (lon_max, lat_max),
    (lon_min, lat_max),
    (lon_min, lat_min),
]
polygon_wgs84 = Polygon(bbox_coords)

# 3. Create GeoDataFrame in WGS84
gdf_wgs84 = gpd.GeoDataFrame(geometry=[polygon_wgs84], crs="EPSG:4326")

# 4. Project to UTM Zone 35N (Egypt) for accurate distance measurements in
meters
gdf_projected = gdf_wgs84.to_crs(epsg=32635)
```

```
# 5. Calculate spatial centroid
centroid_projected = gdf_projected.geometry.centroid.iloc[0]

# 6. FIX: Measure distance from centroid to each corner point vertex
bounding_corners = list(gdf_projected.geometry.iloc[0].exterior.coords)
uncertainty_radius_m = max(
    centroid_projected.distance(Point(pt)) for pt in bounding_corners
)

# 7. Convert Centroid back to WGS84 (Decimal Degrees)
centroid_wgs84 = (
    gpd.GeoSeries([centroid_projected], crs="EPSG:32635")
    .to_crs("EPSG:4326")
    .iloc[0]
)

# Darwin Core Output Format
dwc_output = {
    "decimalLatitude": round(centroid_wgs84.y, 6),
    "decimalLongitude": round(centroid_wgs84.x, 6),
    "coordinateUncertaintyInMeters": int(round(uncertainty_radius_m)),
    "geodeticDatum": "EPSG:4326",
    "georeferenceProtocol": "Point-Radius Method (Bounding Box Centroid)",
}

print("Darwin Core Georeferencing Results:")
for field, val in dwc_output.items():
    print(f"{field}: {val}")
```

## STEP 4: RETRIEVE THE CLIMATIC DATA (TEMPERATURE, RAINFALL, RELATIVE HUMIDITY, WIND SPEED, SOIL TEMPERATURE, SOIL MOISTURE) for plausible area in 1954 and 1964

### Cell#1

```
import pandas as pd
import requests

url = "https://archive-api.open-meteo.com/v1/archive"

CENTER_LAT = 31.202819
CENTER_LON = 27.83665
UNCERTAINTY_METERS = 22309
```

```
# Convert ~22.3 km uncertainty to degrees latitude/longitude
lat_offset = UNCERTAINTY_METERS / 111000  # 1 degree lat ≈ 111 km
lon_offset = UNCERTAINTY_METERS / 95000   # 1 degree lon ≈ 95 km at ~31°N

# Sample center + 4 cardinal points along the uncertainty radius boundary
locations = [
    (CENTER_LAT, CENTER_LON),
    (CENTER_LAT + lat_offset, CENTER_LON),
    (CENTER_LAT - lat_offset, CENTER_LON),
    (CENTER_LAT, CENTER_LON + lon_offset),
    (CENTER_LAT, CENTER_LON - lon_offset),
]

samples = {
    "Sample 1954": ("1945-03-01", "1954-05-31"),
    "Sample 1964": ("1955-06-09", "1964-06-09")
}

def fetch_area_data(sample_name, start_date, end_date):
    # Pass lists of latitudes and longitudes to average across the
uncertainty circle
    lats = [loc[0] for loc in locations]
    lons = [loc[1] for loc in locations]

    params = {
        "latitude": lats,
        "longitude": lons,
        "start_date": start_date,
        "end_date": end_date,
        # Added relative_humidity_2m_mean to daily requested metrics
        "daily": ["temperature_2m_mean", "rain_sum",
"relative_humidity_2m_mean"],
        "timezone": "Africa/Cairo"
    }

    response = requests.get(url, params=params)
    data_list = response.json()

    # If querying multiple locations, Open-Meteo returns a JSON list
    if not isinstance(data_list, list):
        data_list = [data_list]

    dfs = []
    for data in data_list:
        daily = data.get("daily", {})
```

```
        if "time" in daily:
            df = pd.DataFrame({
                "date": pd.to_datetime(daily["time"]),
                "temp": daily["temperature_2m_mean"],
                "rain": daily["rain_sum"],
                "humidity": daily.get("relative_humidity_2m_mean")
            })
            dfs.append(df)

    if dfs:
        # Average weather readings across all sampled points in the radius
        combined_df = pd.concat(dfs).groupby("date").mean()

        # Monthly aggregation: mean temperature, total rainfall sum, mean
relative humidity
        monthly_df = combined_df.resample("ME").agg({
            "temp": "mean",
            "rain": "sum",
            "humidity": "mean"
        }).reset_index()

        print(f"\n--- {sample_name} (Spatially Averaged across
{UNCERTAINTY_METERS}m radius) ---")
        for _, row in monthly_df.iterrows():
            print(f"Date: {row['date'].strftime('%Y-%m')} | Avg Temp:
{row['temp']:.2f}°C | Total Rain: {row['rain']:.2f} mm | Avg Humidity:
{row['humidity']:.2f}%")

        return monthly_df
    return None

# Fetch data for all samples
results = {name: fetch_area_data(name, start, end) for name, (start, end)
in samples.items()}

# ------------------------------------------------------------
# Exporting Results to an Excel File
# ------------------------------------------------------------
output_filename =
"weather_monthly_summary_1_plausible_area_1954_1964.xlsx"

with pd.ExcelWriter(output_filename, engine="openpyxl") as writer:
    for sample_name, df in results.items():
        if df is not None:
            # Format the date column to 'YYYY-MM' before saving to Excel
```

```
            export_df = df.copy()
            export_df["date"] = export_df["date"].dt.strftime("%Y-%m")

            # Write each sample to its own Excel worksheet sheet
            sheet_name = sample_name.replace(":", "")  # Excel sheet names
avoid special characters
            export_df.to_excel(writer, sheet_name=sheet_name, index=False)

print(f"\nSuccessfully exported results to '{output_filename}'.")
```

**Cell#2**

```
import pandas as pd
import requests
import time

url = "https://archive-api.open-meteo.com/v1/archive"

CENTER_LAT = 31.202819
CENTER_LON = 27.83665
UNCERTAINTY_METERS = 22309

# Convert ~22.3 km uncertainty to degrees latitude/longitude
lat_offset = UNCERTAINTY_METERS / 111000  # 1 degree lat ≈ 111 km
lon_offset = UNCERTAINTY_METERS / 95000   # 1 degree lon ≈ 95 km at ~31°N

# Sample center + 4 cardinal points along the uncertainty radius boundary
locations = [
    (CENTER_LAT, CENTER_LON),
    (CENTER_LAT + lat_offset, CENTER_LON),
    (CENTER_LAT - lat_offset, CENTER_LON),
    (CENTER_LAT, CENTER_LON + lon_offset),
    (CENTER_LAT, CENTER_LON - lon_offset),
]

samples = {
    "Sample 1954": ("1945-03-01", "1954-05-31"),
    "Sample 1964": ("1955-06-09", "1964-06-09")
}

def fetch_area_data(sample_name, start_date, end_date):
    print(f"\nFetching data for {sample_name} ({start_date} to
{end_date})...")
    dfs = []
```

```
    # Iterate through locations individually to avoid exceeding API limits
    for idx, (lat, lon) in enumerate(locations, start=1):
        params = {
            "latitude": lat,
            "longitude": lon,
            "start_date": start_date,
            "end_date": end_date,
            # MODIFICATION: Replaced daily metrics with Soil Temp (28-
100cm), Soil Moisture (28-100cm), and Wind Speed (10m)
            "daily": [
                "soil_temperature_28_to_100cm_mean",
                "soil_moisture_28_to_100cm_mean",
                "wind_speed_10m_max"
            ],
            "timezone": "Africa/Cairo"
        }

        # Handle retries for rate limits
        for attempt in range(1, 4):
            try:
                response = requests.get(url, params=params, timeout=45)
                data = response.json()

                if isinstance(data, dict) and data.get("error"):
                    reason = data.get("reason", "")
                    print(f"  ⏳ Rate limit hit on point {idx}. Waiting
65s (Attempt {attempt}/3)...")
                    time.sleep(65)
                    continue

                daily = data.get("daily", {})
                if "time" in daily:
                    df = pd.DataFrame({
                        "date": pd.to_datetime(daily["time"]),
                        "soil_temp_28_100cm":
daily.get("soil_temperature_28_to_100cm_mean"),
                        "soil_moisture_28_100cm":
daily.get("soil_moisture_28_to_100cm_mean"),
                        "wind_speed_10m": daily.get("wind_speed_10m_max")
                    })
                    dfs.append(df)
                break

            except Exception as e:
                print(f"  ❌ Exception fetching point {idx}: {e}")
```

```
                time.sleep(10)

        time.sleep(3)  # Gentle pacing between API calls

    if dfs:
        # Average weather readings across all sampled points in the radius
        combined_df = pd.concat(dfs).groupby("date").mean()

        # Monthly aggregation (Compatible with both new and older pandas
versions)
        resample_rule = "ME" if pd.__version__ >= "2.2.0" else "M"
        monthly_df = combined_df.resample(resample_rule).agg({
            "soil_temp_28_100cm": "mean",
            "soil_moisture_28_100cm": "mean",
            "wind_speed_10m": "mean"
        }).reset_index()

        print(f"\n--- {sample_name} (Spatially Averaged across
{UNCERTAINTY_METERS}m radius) ---")
        for _, row in monthly_df.head(5).iterrows():
            print(
                f"Date: {row['date'].strftime('%Y-%m')} | "
                f"Soil Temp (28-100cm): {row['soil_temp_28_100cm']:.2f}°C
| "
                f"Soil Moisture (28-100cm):
{row['soil_moisture_28_100cm']:.3f} m³/m³ | "
                f"Wind Speed (10m): {row['wind_speed_10m']:.2f} km/h"
            )
        print("... (showing first 5 rows)")

        return monthly_df

    return None

# Fetch data for all samples
results = {name: fetch_area_data(name, start, end) for name, (start, end)
in samples.items()}

# ------------------------------------------------------------
# Exporting Results to an Excel File
# ------------------------------------------------------------
output_filename =
"weather_monthly_summary_2_plausible_area_1954_1964.xlsx"
valid_sheets = 0
```

```
with pd.ExcelWriter(output_filename, engine="openpyxl") as writer:
    for sample_name, df in results.items():
        if df is not None and not df.empty:
            export_df = df.copy()
            export_df["date"] = export_df["date"].dt.strftime("%Y-%m")

            sheet_name = sample_name.replace(":", "")
            export_df.to_excel(writer, sheet_name=sheet_name, index=False)
            valid_sheets += 1

    # Fallback to prevent openpyxl crash if all requests fail
    if valid_sheets == 0:
        pd.DataFrame({"Status": ["No data retrieved. Rate limit
exceeded."]}).to_excel(writer, sheet_name="Error_Log", index=False)

print(f"\nSuccessfully exported results to '{output_filename}'.")
```

## STEP 5: RETRIEVE THE CLIMATIC DATA (TEMPERATURE, RAINFALL, RELATIVE HUMIDITY, WIND SPEED, SOIL TEMPERATURE, SOIL MOISTURE) for precise coordinates in 2006 and 2024

### Cell#1

```
import pandas as pd
import requests

url = "https://archive-api.open-meteo.com/v1/archive"

# Samples with exact coordinates, date ranges, and custom surface
elevations (meters above sea level)
samples = {
    "Sample 2006": {
        "start": "1997-04-29",
        "end": "2006-04-29",
        "lat": 31.238722,  # 31° 14' 19.4" N
        "lon": 27.865111,  # 27° 51' 54.4" E
        "elevation": 20    # Custom target altitude: 20 meters
    },
    "Sample 2024": {
        "start": "2015-05-22",
        "end": "2024-05-22",
        "lat": 31.232508,  # 31° 13' 57.03" N
        "lon": 27.855928,  # 27° 51' 21.34" E
        "elevation": 48    # Custom target altitude: 48 meters
    }
```

```
}

def fetch_point_data(sample_name, config):
    params = {
        "latitude": config["lat"],
        "longitude": config["lon"],
        "elevation": config["elevation"],  # Adjusts atmospheric
reanalysis to specified surface height
        "start_date": config["start"],
        "end_date": config["end"],
        "daily": ["temperature_2m_mean", "rain_sum",
"relative_humidity_2m_mean"],
        "timezone": "Africa/Cairo"
    }

    response = requests.get(url, params=params)
    data = response.json()

    daily = data.get("daily", {})
    if "time" in daily:
        df = pd.DataFrame({
            "date": pd.to_datetime(daily["time"]),
            "temp": daily["temperature_2m_mean"],
            "rain": daily["rain_sum"],
            "humidity": daily.get("relative_humidity_2m_mean")
        })

        # Resample daily readings to monthly aggregates
        monthly_df = df.set_index("date").resample("ME").agg({
            "temp": "mean",
            "rain": "sum",
            "humidity": "mean"
        }).reset_index()

        print(f"\n--- {sample_name} (Location: {config['lat']},
{config['lon']} | Elevation: {config['elevation']}m) ---")
        for _, row in monthly_df.iterrows():
            print(f"Date: {row['date'].strftime('%Y-%m')} | Avg Temp:
{row['temp']:.2f}°C | Total Rain: {row['rain']:.2f} mm | Avg Humidity:
{row['humidity']:.2f}%")

        return monthly_df

    return None
```

```
# Fetch data for all samples
results = {name: fetch_point_data(name, config) for name, config in
samples.items()}

# Exporting Results to an Excel File
output_filename =
"weather_monthly_summary_1_exact_coordinates_2006_2024.xlsx"

with pd.ExcelWriter(output_filename, engine="openpyxl") as writer:
    for sample_name, df in results.items():
        if df is not None:
            export_df = df.copy()
            export_df["date"] = export_df["date"].dt.strftime("%Y-%m")
            sheet_name = sample_name.replace(":", "")
            export_df.to_excel(writer, sheet_name=sheet_name, index=False)

print(f"\nSuccessfully exported results to '{output_filename}'.")
```

**Cell#2**

```
import pandas as pd
import requests
import time

url = "https://archive-api.open-meteo.com/v1/archive"

# Samples with exact coordinates, date ranges, and custom surface
elevations (meters above sea level)
samples = {
    "Sample 2006": {
        "start": "1997-04-29",
        "end": "2006-04-29",
        "lat": 31.238722,   # 31° 14' 19.4" N
        "lon": 27.865111,   # 27° 51' 54.4" E
        "elevation": 20     # Custom target altitude: 20 meters
    },
    "Sample 2024": {
        "start": "2015-05-22",
        "end": "2024-05-22",
        "lat": 31.232508,   # 31° 13' 57.03" N
        "lon": 27.855928,   # 27° 51' 21.34" E
        "elevation": 48     # Custom target altitude: 48 meters
    }
}
```

```
def fetch_point_data(sample_name, config):
    print(f"\nFetching data for {sample_name} ({config['start']} to
{config['end']})...")

    params = {
        "latitude": config["lat"],
        "longitude": config["lon"],
        "elevation": config["elevation"],  # Adjusts atmospheric
reanalysis to specified surface height
        "start_date": config["start"],
        "end_date": config["end"],
        # MODIFICATION: Replaced daily metrics with Soil Temp (28-100cm),
Soil Moisture (28-100cm), and Wind Speed (10m)
        "daily": [
            "soil_temperature_28_to_100cm_mean",
            "soil_moisture_28_to_100cm_mean",
            "wind_speed_10m_max"
        ],
        "timezone": "Africa/Cairo"
    }

    # Handling retries for rate limits
    for attempt in range(1, 4):
        try:
            response = requests.get(url, params=params, timeout=45)
            data = response.json()

            if isinstance(data, dict) and data.get("error"):
                reason = data.get("reason", "")
                print(f"  ⏳ Rate limit hit. Waiting 65s before retry
(Attempt {attempt}/3)...")
                time.sleep(65)
                continue

            daily = data.get("daily", {})
            if "time" in daily:
                df = pd.DataFrame({
                    "date": pd.to_datetime(daily["time"]),
                    "soil_temp_28_100cm":
daily.get("soil_temperature_28_to_100cm_mean"),
                    "soil_moisture_28_100cm":
daily.get("soil_moisture_28_to_100cm_mean"),
                    "wind_speed_10m": daily.get("wind_speed_10m_max")
                })
```

```
                # Monthly aggregation (Compatible with both new and older
pandas versions)
                resample_rule = "ME" if pd.__version__ >= "2.2.0" else "M"
                monthly_df =
df.set_index("date").resample(resample_rule).agg({
                    "soil_temp_28_100cm": "mean",
                    "soil_moisture_28_100cm": "mean",
                    "wind_speed_10m": "mean"
                }).reset_index()

                print(f"--- {sample_name} (Location: {config['lat']},
{config['lon']} | Elevation: {config['elevation']}m) ---")
                for _, row in monthly_df.head(5).iterrows():
                    print(
                        f"Date: {row['date'].strftime('%Y-%m')} | "
                        f"Soil Temp (28-100cm):
{row['soil_temp_28_100cm']:.2f}°C | "
                        f"Soil Moisture (28-100cm):
{row['soil_moisture_28_100cm']:.3f} m³/m³ | "
                        f"Wind Speed (10m): {row['wind_speed_10m']:.2f}
km/h"
                    )
                print("... (showing first 5 rows)")

                return monthly_df
            break

        except Exception as e:
            print(f"  ❌ Exception during request: {e}")
            time.sleep(10)

    return None

# Fetch data for all samples with pacing
results = {}
for name, config in samples.items():
    results[name] = fetch_point_data(name, config)
    time.sleep(3)  # Gentle pacing between requests

# Exporting Results to an Excel File
output_filename =
"weather_monthly_summary_2_exact_coordinates_2006_2024.xlsx"
valid_sheets = 0

with pd.ExcelWriter(output_filename, engine="openpyxl") as writer:
```

```
    for sample_name, df in results.items():
        if df is not None and not df.empty:
            export_df = df.copy()
            export_df["date"] = export_df["date"].dt.strftime("%Y-%m")
            sheet_name = sample_name.replace(":", "")
            export_df.to_excel(writer, sheet_name=sheet_name, index=False)
            valid_sheets += 1

    # Fallback to prevent openpyxl crash if all requests fail
    if valid_sheets == 0:
        pd.DataFrame({"Status": ["No data retrieved. Rate limit
exceeded."]}).to_excel(writer, sheet_name="Error_Log", index=False)

print(f"\nSuccessfully exported results to '{output_filename}'.")
```

## STEP 6: STATISTICS OF PLANT ANATOMY TRAITS

```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy import stats
import seaborn as sns

# Suppress standard warnings
warnings.filterwarnings("ignore")

# Attempt to import scikit-posthocs for Dunn's test (non-parametric)
try:
    import scikit_posthocs as sp
except ImportError:
    import subprocess
    import sys
    subprocess.check_call([sys.executable, "-m", "pip", "install",
"scikit-posthocs", "-q"])
    import scikit_posthocs as sp

# ------------------------------------------------------------
# HELPER FUNCTIONS FOR AUTOMATED POST-HOC LETTER ASSIGNMENTS
# ------------------------------------------------------------
def get_dunn_bonferroni_letters(df_data, val_col, group_col, alpha=0.05):
    """
    Computes Dunn's pairwise post-hoc test with Bonferroni-adjusted p-
values
    and constructs compact letter displays ('a', 'b', 'ab').
```

```
    """
    df_temp = df_data.copy()
    df_temp['rank'] = df_temp[val_col].rank(method='average')
    mean_ranks =
df_temp.groupby(group_col)['rank'].mean().sort_values(ascending=False)
    sorted_groups = mean_ranks.index.tolist()

    # Run Dunn's test with Bonferroni correction
    dunn_p_matrix = sp.posthoc_dunn(df_temp, val_col=val_col,
group_col=group_col, p_adjust='bonferroni')

    letters = {}
    curr_char_code = 97  # ASCII 'a'

    for g1 in sorted_groups:
        if g1 not in letters:
            letters[g1] = chr(curr_char_code)
            for g2 in sorted_groups:
                if g1 != g2 and dunn_p_matrix.loc[g1, g2] >= alpha:
                    if g2 not in letters:
                        letters[g2] = chr(curr_char_code)
            curr_char_code += 1

    return letters

def get_duncan_letters(df_data, val_col, group_col, alpha=0.05):
    """
    Computes Duncan's Multiple Range Test for One-Way ANOVA and returns
compact letters.
    """
    grouped = df_data.groupby(group_col)[val_col]
    means = grouped.mean().sort_values(ascending=False)
    counts = grouped.count()

    k = len(means)
    if k < 2:
        return {g: 'a' for g in means.index}

    sse = sum(grouped.apply(lambda x: np.sum((x - x.mean())**2)))
    n_total = len(df_data[val_col].dropna())
    df_e = n_total - k
    mse = sse / df_e if df_e > 0 else np.var(df_data[val_col])

    n_h = stats.hmean(counts.values)
    se = np.sqrt(mse / n_h)
```

```
    critical_ranges = {}
    for p in range(2, k + 1):
        alpha_p = 1.0 - (1.0 - alpha)**(p - 1)
        q_crit = stats.studentized_range.ppf(1.0 - alpha_p, p, df_e)
        critical_ranges[p] = q_crit * se

    ordered_groups = list(means.index)
    subsets = []

    for i in range(k):
        for j in range(k - 1, i - 1, -1):
            sub = ordered_groups[i : j + 1]
            p_step = len(sub)

            diff = 0.0 if p_step == 1 else abs(means[ordered_groups[i]] -
means[ordered_groups[j]])
            threshold = critical_ranges.get(p_step, 0.0)

            if diff <= threshold:
                if not any(set(sub).issubset(set(existing)) for existing
in subsets):
                    subsets.append(sub)
                break

    group_letters = {g: "" for g in means.index}
    for subset_idx, sub in enumerate(subsets):
        letter = chr(97 + subset_idx)
        for group in sub:
            group_letters[group] += letter

    return group_letters

# ------------------------------------------------------------
# CONFIGURATION & CONTROL PANEL
# ------------------------------------------------------------
FILE_PATH = "/content/drive/MyDrive/final anatomy parameters.xlsx"
LABEL_ROTATION_ANGLE = 50  # Custom rotation angle for header text
annotations
LIGHT_GREEN = "#90EE90"

# Explicitly define parameters to be drawn as BAR PLOTS (Mean ± SD)
BAR_PLOT_PARAMETERS = {"LBT", "AdIBA", "XCVBA", "XDCVB", "TCVBA", "XCHC"}

# Manual post-hoc letter overrides for specific parameters
```

```
CUSTOM_LETTERS = {
    "AbIBA": ["ab", "ab", "a", "b"],
    "TIBA":  ["ab", "ab", "a", "b"],
    "SCVBA": ["ab", "ab", "a", "b"],
    "PCVBA": ["b", "ab", "a", "b"]
}

# Sheet configurations: Label prefix, sheet name, and unit for y-axis
sheet_configs = [
    {"label": "A", "sheet": "LBT", "unit": "(μm)"},
    {"label": "B", "sheet": "AdIBA", "unit": "(μm²)"},
    {"label": "C", "sheet": "AbIBA", "unit": "(μm²)"},
    {"label": "D", "sheet": "TIBA", "unit": "(μm²)"},
    {"label": "E", "sheet": "SCVBA", "unit": "(μm²)"},
    {"label": "F", "sheet": "PCVBA", "unit": "(μm²)"},
    {"label": "G", "sheet": "XCVBA", "unit": "(μm²)"},
    {"label": "H", "sheet": "XDCVB", "unit": "(μm)"},
    {"label": "I", "sheet": "TCVBA", "unit": "(μm²)"},
    {"label": "J", "sheet": "XCHC", "unit": "(×10⁻¹³ μm⁴/MPa)"},
]

# ------------------------------------------------------------
# FIGURE SET-UP (4 rows x 3 columns)
# ------------------------------------------------------------
sns.set_theme(style="white")
fig, axes = plt.subplots(4, 3, figsize=(18, 24), dpi=300)
axes_flat = axes.flatten()

# Hide any extra unused axes
for i in range(len(sheet_configs), len(axes_flat)):
    fig.delaxes(axes_flat[i])

print("=" * 80)
print("PROCESSING EXCEL SHEETS & GENERATING PLOTS WITH CUSTOM/STATISTICAL
POST-HOC")
print("=" * 80)

for ax, config in zip(axes_flat[:len(sheet_configs)], sheet_configs):
    lbl = config["label"]
    s_name = config["sheet"]
    unit = config["unit"]

    param_title = f"{lbl}: {s_name}"
    y_axis_label = f"{s_name} {unit}"
```

```
    # Determine plot type based on specified parameter names
    use_bar_plot = s_name in BAR_PLOT_PARAMETERS

    # 1. Load sheet data
    try:
        df = pd.read_excel(FILE_PATH, sheet_name=s_name, header=0,
engine="openpyxl")
        df.columns = df.columns.astype(str)
    except Exception as e:
        print(f"[{s_name}] Error loading sheet: {e}")
        ax.set_title(f"{param_title}\n(Sheet Missing)", fontsize=10,
fontweight="bold", pad=30)
        ax.set_ylabel(y_axis_label, fontsize=9, fontweight="bold")
        continue

    # Clean data structure
    df = df.dropna(how="all", axis=1).dropna(how="all", axis=0)
    df = df.loc[:, ~df.columns.str.contains("^Unnamed", na=False)]

    for col in df.columns:
        if df[col].dtype == object:
            cleaned_strs = df[col].astype(str).str.replace(r"[^0-9eE\.\+\-
]", "", regex=True)
            df[col] = pd.to_numeric(cleaned_strs, errors="coerce")
        else:
            df[col] = pd.to_numeric(df[col], errors="coerce")

    column_order = [str(c) for c in df.columns]
    df_melted = df.melt(var_name="Group", value_name="Value").dropna()
    df_melted["Group"] = df_melted["Group"].astype(str)

    if df_melted.empty:
        print(f"[{s_name}] Sheet is empty after cleaning.")
        ax.set_title(f"{param_title}\n(No Data)", fontsize=10,
fontweight="bold", pad=30)
        ax.set_ylabel(y_axis_label, fontsize=9, fontweight="bold")
        continue

    # Keep absolute raw unscaled values isolated
    df_melted["Raw_Value"] = df_melted["Value"].copy()

    # 2. Determine Scale Factor for Y-Axis
    max_val = df_melted["Value"].abs().max()
    scale_factor = 1.0  # Default: 1:1 scaling
```

```
    if s_name == "XCHC":
        if max_val != 0 and (max_val < 1e-3 or max_val > 1e3):
            exponent = -13 if max_val < 1 else
int(np.floor(np.log10(max_val)))
            scale_factor = 10**exponent
            clean_unit = "μm⁴/MPa"
            y_axis_label = f"{s_name} (×10⁻¹³ {clean_unit})"
    elif max_val >= 1000:
        exponent = int(np.floor(np.log10(max_val)))
        scale_factor = 10**exponent
        clean_unit = unit.strip("()")
        y_axis_label = f"{s_name} (×10^{exponent} {clean_unit})"

    # Apply scaling to plot array
    df_melted["Value"] = df_melted["Raw_Value"] / scale_factor

    # 3. Statistical Testing
    raw_vals = df_melted["Raw_Value"].values
    shapiro_stat_raw, shapiro_p_raw = stats.shapiro(raw_vals) if
len(raw_vals) >= 3 else (np.nan, 0.0)
    is_normal = shapiro_p_raw >= 0.05

    groups_list = [df_melted[df_melted["Group"] ==
grp]["Raw_Value"].values for grp in column_order]

    if is_normal:
        test_type = "ANOVA"
        stat, p_val = stats.f_oneway(*groups_list)
    else:
        test_type = "Kruskal-Wallis"
        stat, p_val = stats.kruskal(*groups_list)

    # 4. Assign Post-Hoc Letters (Check Custom Mapping First)
    if s_name in CUSTOM_LETTERS:
        custom_list = CUSTOM_LETTERS[s_name]
        letters_dict = {
            grp: custom_list[idx] if idx < len(custom_list) else 'a'
            for idx, grp in enumerate(column_order)
        }
    else:
        letters_dict = {grp: 'a' for grp in column_order}
        if p_val < 0.05:
            try:
                if is_normal:
```

```
                    letters_dict = get_duncan_letters(df_melted,
val_col="Raw_Value", group_col="Group", alpha=0.05)
                else:
                    letters_dict = get_dunn_bonferroni_letters(df_melted,
val_col="Raw_Value", group_col="Group", alpha=0.05)
            except Exception as e:
                print(f"[{s_name}] Post-hoc calculation error: {e}.
Defaulting to 'a'.")

    # ---------------------------------------------------------
    # DRAW PLOTS BASED ON PARAMETER SPECIFICATION
    # ---------------------------------------------------------
    if use_bar_plot:
        # BAR PLOT with Mean ± Standard Deviation (SD)
        sns.barplot(
            data=df_melted, x="Group", y="Value", order=column_order,
ax=ax,
            color=LIGHT_GREEN, edgecolor="#333333", linewidth=1.2,
capsize=0.1, errorbar="sd"
        )
        plot_tag = "[Bar: Mean ± SD]"
    else:
        # VIOLIN PLOT (Median & IQR)
        for group in df_melted["Group"].unique():
            mask = df_melted["Group"] == group
            if mask.sum() > 0 and df_melted.loc[mask, "Value"].std() == 0:
                df_melted.loc[mask, "Value"] += np.random.normal(0, 1e-6,
size=mask.sum())

        sns.violinplot(
            data=df_melted, x="Group", y="Value", order=column_order,
ax=ax,
            color=LIGHT_GREEN, inner="quartile", cut=0, bw_adjust=0.5
        )
        plot_tag = "[Violin Plot]"

    # Title and Axes Labels
    ax.set_title(f"{param_title} {plot_tag}\n{test_type} (p =
{p_val:.4f})", fontsize=10, fontweight="bold", pad=70)
    ax.set_xlabel("")
    ax.set_ylabel(y_axis_label, fontsize=9, fontweight="bold")
    ax.set_xticklabels(ax.get_xticklabels(), rotation=30, ha="right",
rotation_mode="anchor", fontsize=8.5, fontweight="bold")

    # Expand Y-limits slightly for letter and header annotations
```

```
    y_min, y_max = ax.get_ylim()
    ax.set_ylim(y_min, y_max * 1.18)

    # ------------------------------------------------------------
    # ANNOTATE ACCURATE STATISTICS & COMPACT LETTERS
    # ------------------------------------------------------------
    for x_idx, col in enumerate(column_order):
        col_vals_raw = df_melted[df_melted["Group"] ==
col]["Raw_Value"].dropna()
        col_vals_scaled = df_melted[df_melted["Group"] ==
col]["Value"].dropna()

        if col_vals_raw.empty:
            continue

        letter = letters_dict.get(col, 'a')

        if use_bar_plot:
            # Stats for Bar Plot: Calculate Mean & SD on Raw Data then
apply Scale Factor
            raw_mean = col_vals_raw.mean()
            raw_sd = col_vals_raw.std() if len(col_vals_raw) > 1 else 0.0

            mean_val = raw_mean / scale_factor
            sd_val = raw_sd / scale_factor

            fmt = ".2e" if abs(mean_val) < 0.01 and mean_val != 0 else
".2f"
            header_stat_text = f"Mean = {mean_val:{fmt}}\nSD =
{sd_val:{fmt}}"
            letter_y_pos = mean_val + sd_val + (y_max * 0.03)
        else:
            # Stats for Violin Plot: Calculate Median & IQR directly on
Raw Data, then Scale
            raw_median = col_vals_raw.median()
            q75_raw, q25_raw = np.percentile(col_vals_raw, [75, 25])
            raw_iqr = q75_raw - q25_raw

            med_val = raw_median / scale_factor
            iqr_val = raw_iqr / scale_factor

            fmt = ".2e" if abs(med_val) < 0.01 and med_val != 0 else ".2f"
            header_stat_text = f"Median = {med_val:{fmt}}\nIQR =
{iqr_val:{fmt}}"
            letter_y_pos = col_vals_scaled.max() + (y_max * 0.03)
```

```
        # 1. Place Post-Hoc Compact Letter on top of Bar/Violin
        ax.text(
            x=x_idx, y=letter_y_pos, s=letter,
            ha="center", va="bottom",
            fontsize=10.5, fontweight="bold", color="black"
        )

        # 2. Place Statistics text rotated above the subplot top border
        ax.text(
            x=x_idx, y=1.02, s=header_stat_text,
transform=ax.get_xaxis_transform(),
            ha="left", va="bottom", rotation=LABEL_ROTATION_ANGLE,
rotation_mode="anchor",
            fontsize=7, fontweight="bold", linespacing=1.1
        )

    ax.grid(False)

# Final Layout Adjustments & Save Plot Image
plt.subplots_adjust(wspace=0.32, hspace=0.60, top=0.93, bottom=0.05,
left=0.06, right=0.96)
output_img_path = "/content/anatomy_parameters_bar_and_violin_plots.png"
plt.savefig(output_img_path, dpi=300, bbox_inches="tight")
plt.show()

print(f"\nPlot generation complete! Saved image to: {output_img_path}")
```

**STEP 7: STATISTICS OF CLIMATE PARAMETERS**

```
%pip install scikit-posthocs -q

from google.colab import drive, files
import pandas as pd
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns
import scikit_posthocs as sp

# Publication styling settings
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.sans-serif'] = ['Arial', 'DejaVu Sans']
plt.rcParams['pdf.fonttype'] = 42  # Vector fonts for publication
editability
```

```
sns.set_theme(style="white")

# 1. Mount Google Drive
drive.mount('/content/drive')

# 2. File path and parameter definitions
file_path = "/content/drive/MyDrive/ALL CLIMATE.xlsx"
parameters = [
    'temp',
    'rain',
    'humidity',
    'soil_temp_28_100cm',
    'soil_moisture_28_100cm',
    'wind_speed_10m'
]

# Uppercase letters for subplot panel tagging (A, B, C, D, E, F...)
panel_labels = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

# Helper function to generate post-hoc compact letter displays matching
SPSS Bonferroni-Dunn's test
def get_spss_dunn_bonferroni_letters(param_df, val_col, group_col,
alpha=0.05):
    """
    Replicates SPSS pairwise post-hoc test after Kruskal-Wallis:
    1. Computes mean ranks for each group.
    2. Runs Dunn's pairwise test using Bonferroni-adjusted p-values.
    3. Groups non-significantly different groups into compact letter
displays ('a', 'b', 'ab').
    """
    param_df = param_df.copy()

    # 1. Rank data overall across all groups
    param_df['rank'] = param_df[val_col].rank(method='average')

    # 2. Calculate Mean Ranks for sorting groups
    mean_ranks =
param_df.groupby(group_col)['rank'].mean().sort_values(ascending=False)
    sorted_groups = mean_ranks.index.tolist()

    # 3. Perform Dunn's pairwise post-hoc test with Bonferroni adjustment
    dunn_p_matrix = sp.posthoc_dunn(param_df, val_col=val_col,
group_col=group_col, p_adjust='bonferroni')
```

```
    # 4. Construct compact letter display based on adjusted pairwise p-
values and mean ranks
    letters = {}
    curr_char_code = 97  # ASCII 'a'

    for g1 in sorted_groups:
        if g1 not in letters:
            letters[g1] = chr(curr_char_code)
            for g2 in sorted_groups:
                if g1 != g2 and dunn_p_matrix.loc[g1, g2] >= alpha:
                    if g2 not in letters:
                        letters[g2] = chr(curr_char_code)
            curr_char_code += 1

    return letters

# Helper function to compute SPSS-Exact Duncan's Multiple Range Test
Letters
def get_spss_duncan_letters(param_df, param_col, group_col, alpha=0.05):
    """
    Computes Duncan's Multiple Range Test:
    1. Ranks group means.
    2. Calculates Mean Squared Error (MSE) & Pooled Error Degrees of
Freedom (df_e).
    3. Uses Studentized Range q-critical values for steps p = 2..k.
    4. Constructs homogeneous subsets and outputs compact letter displays
('a', 'ab', 'b').
    """
    grouped = param_df.groupby(group_col)[param_col]
    means = grouped.mean().sort_values(ascending=False)
    counts = grouped.count()

    k = len(means)
    if k < 2:
        return {g: 'a' for g in means.index}

    # Calculate One-Way ANOVA MSE and df_error
    sse = sum(grouped.apply(lambda x: np.sum((x - x.mean())**2)))
    n_total = len(param_df[param_col].dropna())
    df_e = n_total - k
    mse = sse / df_e if df_e > 0 else np.var(param_df[param_col])

    # Harmonic mean for sample size
    n_h = stats.hmean(counts.values)
    se = np.sqrt(mse / n_h)
```

```
    # Calculate Critical Ranges (R_p) for steps p = 2...k
    critical_ranges = {}
    for p in range(2, k + 1):
        alpha_p = 1.0 - (1.0 - alpha)**(p - 1)
        q_crit = stats.studentized_range.ppf(1.0 - alpha_p, p, df_e)
        critical_ranges[p] = q_crit * se

    # Identify homogeneous subsets
    ordered_groups = list(means.index)
    subsets = []

    for i in range(k):
        for j in range(k - 1, i - 1, -1):
            sub = ordered_groups[i : j + 1]
            p_step = len(sub)

            if p_step == 1:
                diff = 0.0
            else:
                diff = abs(means[ordered_groups[i]] -
means[ordered_groups[j]])

            threshold = critical_ranges[p_step] if p_step in
critical_ranges else 0.0

            if diff <= threshold:
                if not any(set(sub).issubset(set(existing)) for existing
in subsets):
                    subsets.append(sub)
                break

    # Assign subset letters (Subset 1 -> 'a', Subset 2 -> 'b', etc.)
    group_letters = {g: "" for g in means.index}
    for subset_idx, sub in enumerate(subsets):
        letter = chr(97 + subset_idx)
        for group in sub:
            group_letters[group] += letter

    return group_letters

# 3. Read Excel file with 2 header rows
raw_df = pd.read_excel(file_path, header=[0, 1])

# Handle merged year headers in Excel
```

```
cols = raw_df.columns.to_frame()
cols[0] = cols[0].astype(str).str.replace(r'^Unnamed:.*', '', regex=True)
cols[0] = cols[0].replace('', np.nan).ffill()
raw_df.columns = pd.MultiIndex.from_frame(cols)

# Define explicit mapping from individual year labels to date ranges
year_mapping = {
    '1954': '1945-1954',
    '1964': '1955-1964',
    '2006': '1997-2006',
    '2024': '2015-2024'
}

# Extract unique years present in dataset and map them to date ranges
raw_years = [str(yr).strip() for yr in
raw_df.columns.get_level_values(0).unique() if str(yr).strip() != '']
years = [year_mapping.get(yr, yr) for yr in raw_years]

# 4. Reshape data into long format for analysis using mapped year ranges
long_data_list = []
for orig_yr, mapped_yr in zip(raw_years, years):
    sub_df = raw_df[orig_yr].copy()
    sub_df['Year'] = str(mapped_yr).strip()
    long_data_list.append(sub_df)

df = pd.concat(long_data_list, ignore_index=True)

# 5. Statistical Testing & Combined Figure Generation
normality_summary = []
test_results = []

# Create 1 main figure containing a 2x3 grid of subplots (3 columns x 2
rows)
fig, axes = plt.subplots(2, 3, figsize=(18, 12), dpi=300)
axes = axes.flatten()  # Flatten matrix to 1D array for easy iteration

for idx, param in enumerate(parameters):
    ax = axes[idx]
    ax.grid(False)  # Explicitly disable background grid lines
    panel_label = panel_labels[idx] if idx < len(panel_labels) else chr(65
+ idx)

    if param not in df.columns:
        print(f"Skipping '{param}': Column not found in dataset.")
        ax.set_visible(False)
```

```
            continue

        # Ensure parameter data is clean and numeric
        df[param] = pd.to_numeric(df[param], errors='coerce')
        param_df = df.dropna(subset=[param]).copy()

        # ------------------------------------------------------------------
-----
        # OVERALL NORMALITY ASSESSMENT ACROSS ALL DATA
        # ------------------------------------------------------------------
-----
        all_values = param_df[param]

        if len(all_values) >= 3:
            shapiro_stat, shapiro_p = stats.shapiro(all_values)
            is_overall_normal = shapiro_p > 0.05  # Normal if p > 0.05
        else:
            shapiro_stat, shapiro_p = np.nan, 0.0
            is_overall_normal = False

        normality_summary.append({
            'Parameter': param,
            'Total N': len(all_values),
            'Shapiro W-Stat': round(shapiro_stat, 4),
            'Normality p-value': round(shapiro_p, 5),
            'Overall Normal?': 'Yes' if is_overall_normal else 'No'
        })

        # Prepare yearly group arrays
        grouped_series = {yr: param_df[param_df['Year'] == yr][param] for yr
in years}
        groups = [grouped_series[yr] for yr in years if
len(grouped_series[yr]) > 0]

        if len(groups) < 2:
            print(f"Not enough valid groups to compare '{param}'.")
            ax.set_visible(False)
            continue

        # Execute main statistical comparison based on overall normality
        if is_overall_normal:
            test_type = "One-Way ANOVA"
            stat, p_val = stats.f_oneway(*groups)
        else:
            test_type = "Kruskal-Wallis"
```

```
        stat, p_val = stats.kruskal(*groups)

    test_results.append({
        'Climate Parameter': param,
        'Normality (Overall)': 'Normal' if is_overall_normal else 'Non-
Normal',
        'Selected Test': test_type,
        'p-value': round(p_val, 5),
        'Significance (α=0.05)': 'Significant Difference' if p_val < 0.05
else 'No Significant Difference'
    })

    #
==========================================================================
    # POST-HOC PAIRWISE SIGNIFICANCE LETTER CALCULATIONS
    #
==========================================================================
    letters_dict = {yr: 'a' for yr in years}  # Default if no significant
overall difference

    if p_val < 0.05:
        if is_overall_normal:
            # --- DUNCAN TEST FOR NORMAL DATA ---
            try:
                letters_dict = get_spss_duncan_letters(param_df, param,
'Year', alpha=0.05)
            except Exception as e:
                print(f"[{param}] Duncan test failed: {e}. Falling back to
default lettering.")
                letters_dict = {yr: 'a' for yr in years}

        else:
            # --- DUNN'S TEST WITH BONFERRONI ADJUSTMENT FOR NON-NORMAL
DATA ---
            letters_dict = get_spss_dunn_bonferroni_letters(param_df,
val_col=param, group_col='Year', alpha=0.05)

    #
==========================================================================
    # PLOTTING SECTION (UNIFIED GRAY COLOR, WITH CAPITAL PANEL LETTERS)
    #
==========================================================================
    sig_label = "Significant (p < 0.05)" if p_val < 0.05 else "Not
Significant (p ≥ 0.05)"
```

```
    if is_overall_normal:
        # --- NORMAL DATA: GRAY BAR CHART ---
        stats_per_year = param_df.groupby('Year')[param].agg(
            mean='mean',
            std='std'
        ).reindex(years)

        bars = ax.bar(
            stats_per_year.index,
            stats_per_year['mean'],
            yerr=stats_per_year['std'],
            capsize=5,
            color='#808080',  # Standard Gray
            edgecolor='black',
            linewidth=1.0,
            alpha=0.85
        )

        max_y = (stats_per_year['mean'] + stats_per_year['std']).max()
        ax.set_ylim(0, max_y * 1.35 if max_y > 0 else 1)

        # Annotate stats and significance letters
        for bar, (yr, row) in zip(bars, stats_per_year.iterrows()):
            h = bar.get_height()
            std_val = row['std'] if not np.isnan(row['std']) else 0
            letter = letters_dict.get(yr, 'a')

            # Letter
            ax.text(
                bar.get_x() + bar.get_width() / 2.0,
                h + std_val + (max_y * 0.08),
                letter,
                ha='center', va='bottom', fontsize=12, fontweight='bold',
color='black'
            )
            # Mean & SD
            ax.text(
                bar.get_x() + bar.get_width() / 2.0,
                h + std_val + (max_y * 0.02),
                f"Mean: {row['mean']:.2f}\nSD: ±{std_val:.2f}",
                ha='center', va='bottom', fontsize=8, fontweight='bold',
color='black'
            )
```

```
        chart_title = f"{panel_label}. {param} [NORMAL]\nBar Chart (Mean ± SD) | ANOVA (p = {p_val:.4f})\n[{sig_label}]"

    else:
        # --- NON-NORMAL DATA: GRAY VIOLIN PLOT ---
        sns.violinplot(
            x='Year',
            y=param,
            data=param_df,
            order=years,
            ax=ax,
            color='#C0C0C0',  # Light Gray
            inner="quartile",
            cut=0,
            linewidth=1.0
        )

        stats_per_year = param_df.groupby('Year')[param].agg(
            median='median',
            q25=lambda x: np.percentile(x.dropna(), 25),
            q75=lambda x: np.percentile(x.dropna(), 75)
        ).reindex(years)
        stats_per_year['iqr'] = stats_per_year['q75'] - stats_per_year['q25']

        max_y = param_df[param].max()
        min_y = param_df[param].min()
        data_range = max_y - min_y if max_y != min_y else 1
        ax.set_ylim(min_y - (data_range * 0.1), max_y + (data_range * 0.40))

        # Annotate stats and significance letters
        for letter_idx, yr in enumerate(years):
            row = stats_per_year.loc[yr]
            med = row['median']
            iqr = row['iqr']
            letter = letters_dict.get(yr, 'a')

            # Letter
            ax.text(
                letter_idx,
                max_y + (data_range * 0.18),
                letter,
                ha='center', va='bottom', fontsize=12, fontweight='bold', color='black'
```

```
            )
            # Median & IQR
            ax.text(
                letter_idx,
                max_y + (data_range * 0.04),
                f"Med: {med:.2f}\nIQR: {iqr:.2f}",
                ha='center', va='bottom', fontsize=8, fontweight='bold',
color='black'
            )

        chart_title = f"{panel_label}. {param} [NON-NORMAL]\nViolin Plot
(Median & IQR) | Kruskal-Wallis (p = {p_val:.4f})\n[{sig_label}]"

    ax.set_title(chart_title, fontsize=10, fontweight='bold', pad=10)
    ax.set_xlabel("Time Period", fontsize=10, fontweight='bold')
    ax.set_ylabel(param, fontsize=10, fontweight='bold')

    # Stylize spine borders for journal standards
    for spine in ax.spines.values():
        spine.set_linewidth(1.0)
        spine.set_color('black')

# Adjust layout
plt.tight_layout()

#
================================================================================
# PUBLICATION EXPORT & DOWNLOAD
#
================================================================================
png_filename = "Climate_Parameters_Comparison_300DPI.png"
pdf_filename = "Climate_Parameters_Comparison_Vector.pdf"

# Save figure in high-res PNG (300 DPI) and PDF vector format
fig.savefig(png_filename, format='png', dpi=300, bbox_inches='tight',
facecolor='white')
fig.savefig(pdf_filename, format='pdf', dpi=300, bbox_inches='tight',
facecolor='white')

plt.show()

# Trigger automatic browser download in Colab
print("\nDownloading high-resolution publication figures...")
files.download(png_filename)
files.download(pdf_filename)
```

```
# 6. Print Summary Output Tables
normality_df = pd.DataFrame(normality_summary)
results_df = pd.DataFrame(test_results)

print("\n" + "="*80)
print("1. OVERALL SHAPIRO-WILK NORMALITY TEST SUMMARY (ALL TIME PERIODS
POOLED)")
print("="*80)
print(normality_df.to_string(index=False))

print("\n" + "="*80)
print("2. HYPOTHESIS TEST RESULTS ACROSS TIME PERIODS")
print("="*80)
print(results_df.to_string(index=False))
```

**STEP 8: Model Building**

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression
from sklearn.preprocessing import RobustScaler
from sklearn.metrics import roc_curve, auc,  mean_squared_error
!pip install -q adjustText
from adjustText import adjust_text

#
==========================================================================
====
# 1. LOAD DATA & COMPUTE MEDIANS (CLIMATE & PLANT TRAITS)
#
==========================================================================
====
excel_climate = '/content/drive/MyDrive/ALL CLIMATE.xlsx'
df_clim_raw = pd.read_excel(excel_climate, sheet_name=0)

years = ['1954', '1964', '2006', '2024']
col_groups = [range(0, 7), range(7, 14), range(14, 21), range(21, 28)]
climate_features = ['temp', 'rain', 'humidity', 'soil_temp_28_100cm',
'soil_moisture_28_100cm', 'wind_speed_10m']

# Extract 10-year climate medians per timepoint
```

```
climate_medians = []
for yr, cols in zip(years, col_groups):
    sub = df_clim_raw.iloc[1:, cols].copy()
    sub.columns = df_clim_raw.iloc[0, cols].values
    sub = sub.dropna(how='all')
    for c in climate_features:
        sub[c] = pd.to_numeric(sub[c], errors='coerce')
    meds = sub[climate_features].median()
    meds['Year'] = int(yr)
    climate_medians.append(meds)

df_clim_summary = pd.DataFrame(climate_medians).set_index('Year')

# Perform PCA to extract Climate Indices (PC1 & PC2)
scaler_climate = RobustScaler()
X_clim_scaled =
scaler_climate.fit_transform(df_clim_summary[climate_features])
pca = PCA(n_components=2)
X_pcs = pca.fit_transform(X_clim_scaled)
var_exp = pca.explained_variance_ratio_ * 100
print(f"PC1 Explained Variance: {var_exp[0]:.2f}%")
print(f"PC2 Explained Variance: {var_exp[1]:.2f}%")
print(f"Cumulative Variance (PC1 + PC2): {np.sum(var_exp):.2f}%\n")

# 2. Extract and display PCA loadings (correlations/weights of each
climate variable)
df_loadings = pd.DataFrame(
    pca.components_.T,
    index=climate_features,
    columns=['PC1_Loadings', 'PC2_Loadings']
)

# Add absolute contributions to easily see which variables dominate each
PC
df_loadings['PC1_Abs'] = df_loadings['PC1_Loadings'].abs()
df_loadings['PC2_Abs'] = df_loadings['PC2_Loadings'].abs()

print("PCA Component Loadings:")
print(df_loadings.round(4).to_string())

# Define df_pcs first so it can be printed correctly
df_pcs = pd.DataFrame(X_pcs, index=df_clim_summary.index,
columns=['PC1_Climate', 'PC2_Climate'])

# 3. Also inspect the actual PC scores for each of the 4 years
```

```
print("\nYear-by-Year PC Scores:")
print(df_pcs.round(4).to_string())

# Load Plant Anatomy Parameters across all 10 sub-sheets
excel_anatomy = '/content/drive/MyDrive/final anatomy parameters.xlsx'
xls_anat = pd.ExcelFile(excel_anatomy)
plant_traits = xls_anat.sheet_names

pooled_samples = []
for trait in plant_traits:
    df_t = pd.read_excel(excel_anatomy, sheet_name=trait)
    for yr in [1954, 1964, 2006, 2024]:
        vals = pd.to_numeric(df_t[yr], errors='coerce').dropna().values
        pc1_val = df_pcs.loc[int(yr), 'PC1_Climate']
        pc2_val = df_pcs.loc[int(yr), 'PC2_Climate']
        for v in vals:
            pooled_samples.append({'Trait': trait, 'Year': yr, 'PC1':
pc1_val, 'PC2': pc2_val, 'Value': v})

df_all_samples = pd.DataFrame(pooled_samples)

#
================================================================================
====
# 2. PLSR NATIVE FACTOR REDUCTION & VIP CALCULATION
#
================================================================================
====
def calculate_vip(model):
    t = model.x_scores_
    w = model.x_weights_
    q = model.y_loadings_
    p, h = w.shape
    vips = np.zeros((p,))
    s = np.diag(t.T @ t @ q.T @ q)
    total_s = np.sum(s)
    for i in range(p):
        weight = np.zeros((h,))
        for k in range(h):
            weight[k] = (w[i, k] / np.linalg.norm(w[:, k])) ** 2
        vips[i] = np.sqrt(p * np.sum(s * weight) / total_s)
    return vips

from sklearn.cross_decomposition import PLSRegression
from sklearn.preprocessing import RobustScaler
```

```
from sklearn.metrics import roc_curve, auc, mean_squared_error
import numpy as np
import pandas as pd

master_metrics = []

for trait in plant_traits:
    sub = df_all_samples[df_all_samples['Trait'] == trait].copy()
    X_sub = sub[['PC1', 'PC2']]

    # 1. Scale continuous target (median = 0, IQR = 1) & create binary
median-split target
    scaler_y = RobustScaler()
    y_cont = scaler_y.fit_transform(sub[['Value']].values).ravel()
    y_binary = (y_cont > 0).astype(int)

    # 2. Fit PLS-DA Model (Binary Median-Split Target)
    pls_da = PLSRegression(n_components=2, scale=False)
    pls_da.fit(X_sub, y_binary)
    da_scores = pls_da.predict(X_sub).ravel()
    da_vips = calculate_vip(pls_da)
    da_coefs = pls_da.coef_.ravel()
    r2_da = pls_da.score(X_sub, y_binary)

    # ROC-AUC & Youden's J optimal Sensitivity / Specificity
    fpr, tpr, _ = roc_curve(y_binary, da_scores)
    roc_auc = auc(fpr, tpr)
    best_idx = np.argmax(tpr - fpr)

    # 3. Fit Continuous PLSR Model (Full Continuous Target for Robustness
Check)
    pls_reg = PLSRegression(n_components=2, scale=False)
    pls_reg.fit(X_sub, y_cont)
    reg_preds = pls_reg.predict(X_sub).ravel()
    reg_vips = calculate_vip(pls_reg)
    r2_cont = pls_reg.score(X_sub, y_cont)
    rmse_cont = np.sqrt(mean_squared_error(y_cont, reg_preds))

    # Dominant component based on VIP score
    dom_index = 'PC1 (Thermal)' if da_vips[0] > da_vips[1] else 'PC2
(Hydro-Climatic)'

    master_metrics.append({
        'Trait': trait,
        'Median': round(float(scaler_y.center_[0]), 4),
```

```
        'IQR': round(float(scaler_y.scale_[0]), 4),
        # PLS-DA Classification & Discrimination Metrics
        'AUC': round(float(roc_auc), 4),
        'Sensitivity': round(float(tpr[best_idx]), 4),
        'Specificity': round(float(1 - fpr[best_idx]), 4),
        'PLSDA_R2Y': round(float(r2_da), 4),
        'PC1_VIP': round(float(da_vips[0]), 4),
        'PC2_VIP': round(float(da_vips[1]), 4),
        'Max_VIP': round(float(np.max(da_vips)), 4),
        'Dominant_Index': dom_index,
        # PLS-DA Latent Weights & Regression Coefficients
        'PC1_Weight_LV1': round(float(pls_da.x_weights_[0, 0]), 4),
        'PC2_Weight_LV1': round(float(pls_da.x_weights_[1, 0]), 4),
        'PC1_Weight_LV2': round(float(pls_da.x_weights_[0, 1]), 4),
        'PC2_Weight_LV2': round(float(pls_da.x_weights_[1, 1]), 4),
        'PC1_Coef': round(float(da_coefs[0]), 4),
        'PC2_Coef': round(float(da_coefs[1]), 4),
        # Continuous PLSR Validation Metrics
        'PLSR_R2Y': round(float(r2_cont), 4),
        'PLSR_RMSE': round(float(rmse_cont), 4),
        'PLSR_PC1_VIP': round(float(reg_vips[0]), 4),
        'PLSR_PC2_VIP': round(float(reg_vips[1]), 4),
        'VIP_Agreement': bool((da_vips[0] > da_vips[1]) == (reg_vips[0] >
reg_vips[1]))
    })

# Create single unified DataFrame
df_metrics_master = pd.DataFrame(master_metrics)
print("Unified PCA-PLS-DA & PLSR Comparison Matrix:")
print(df_metrics_master.to_string(index=False))

# Export unified workbook
df_metrics_master.to_excel('/content/drive/MyDrive/Unified_PLSR_PLSDA_Metr
ics.xlsx', index=False)

# Keep alias for downstream dumbbell plotting script and trait ordering
df_metrics_AUC = df_metrics_master.copy()
traits = df_metrics_master['Trait'].tolist()

#
================================================================================
====
# PLOT 1: Climate Index Impact / Importance Heatmap Across 10 Plant Traits
```

```
#
================================================================================
====

df_plot = df_metrics_master.sort_values('PLSR_R2Y',
ascending=True).reset_index(drop=True)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(11, 6), sharey=True)

# Left Panel: Explained Variance (PLSR_R2Y)
ax1.hlines(y=df_plot['Trait'], xmin=0, xmax=df_plot['PLSR_R2Y'],
color='#2b5c8f', alpha=0.6, linewidth=2.5)
ax1.scatter(df_plot['PLSR_R2Y'], df_plot['Trait'], color='#2b5c8f', s=120,
zorder=3)
ax1.axvline(0.50, color='gray', linestyle='--', linewidth=1.1,
label='Moderate-Strong Fit (R²Y = 0.50)')
ax1.set_xlabel('Explained Trait Variance (PLSR R²Y)', fontsize=11,
fontweight='bold')
ax1.set_ylabel('Anatomical Trait', fontsize=11, fontweight='bold')
ax1.set_title('Model Explanatory Power (R²Y)', fontsize=12,
fontweight='bold')
ax1.set_xlim(0, max(0.85, df_plot['PLSR_R2Y'].max() * 1.15))
ax1.grid(True, linestyle='--', alpha=0.4)
ax1.legend(loc='lower right', fontsize=9)

# Right Panel: Prediction Error (PLSR_RMSE in IQR units)
ax2.hlines(y=df_plot['Trait'], xmin=0, xmax=df_plot['PLSR_RMSE'],
color='#d95f02', alpha=0.6, linewidth=2.5)
ax2.scatter(df_plot['PLSR_RMSE'], df_plot['Trait'], color='#d95f02',
s=120, marker='s', zorder=3)
ax2.axvline(1.0, color='gray', linestyle=':', linewidth=1.1, label='1.0
IQR Baseline Error')
ax2.set_xlabel('Standardized Prediction Error (RMSE, IQR units)',
fontsize=11, fontweight='bold')
ax2.set_title('Model Prediction Error (RMSE)', fontsize=12,
fontweight='bold')
ax2.set_xlim(0, max(1.2, df_plot['PLSR_RMSE'].max() * 1.15))
ax2.grid(True, linestyle='--', alpha=0.4)
ax2.legend(loc='lower right', fontsize=9)

plt.suptitle('Continuous PLSR Model Performance Across Anatomical Traits',
fontsize=13.5, fontweight='bold', y=1.02)
plt.tight_layout()
plt.savefig('plsr_r2y_rmse_comparison.png', dpi=300, bbox_inches='tight')
plt.show()
```

```
##################
coef_matrix = pd.DataFrame(
    [df_metrics_master['PC1_Coef'].values,
df_metrics_master['PC2_Coef'].values],
    index=['PC1 Index (Temp/Soil Temp)', 'PC2 Index (Rain/Humidity)'],
    columns=traits
)

fig, ax = plt.subplots(figsize=(12, 6.5))

sns.heatmap(
    coef_matrix,
    annot=True,
    fmt='.3f',
    cmap='vlag',          # Diverging blue-white-red palette matching your
figure
    center=0,
    annot_kws={'size': 12},
    cbar_kws={'label': 'PLSR Effect Coefficient'},
    ax=ax
)

ax.set_title('Climate Index Effect Coeffecoent Heatmap Across 10 Plant
Traits',
             fontsize=15, fontweight='bold', pad=12)
ax.set_yticklabels(ax.get_yticklabels(), rotation=0, fontsize=11,
fontweight='bold')
ax.set_xticklabels(ax.get_xticklabels(), rotation=30, ha='right',
fontsize=11, fontweight='bold')

# Style colorbar label
cbar = ax.collections[0].colorbar
cbar.ax.tick_params(labelsize=11)
cbar.set_label('PLSR Effect Coefficient', fontsize=12)

plt.tight_layout()
plt.savefig('climate_index_impact_heatmap.png', dpi=300)
plt.show()

#
==========================================================================
====
```

```
# PLOT 2: PLSR Variable Importance in Projection (VIP) per Plant Trait in
Each model
#
================================================================================
====
x = np.arange(len(traits))
width = 0.38

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6), sharey=True)

# ----------------------------------------------------------
# Panel A: PLS-DA VIP Scores (Binary Median-Split Model)
# ----------------------------------------------------------
ax1.bar(x - width / 2, df_metrics_master['PC1_VIP'], width,
        label='Climate Index 1 (PC1: Thermal)', color='#28598a')
ax1.bar(x + width / 2, df_metrics_master['PC2_VIP'], width,
        label='Climate Index 2 (PC2: Moisture/Rain)', color='#d95f02')
ax1.axhline(1.0, color='red', linestyle='--', linewidth=1.6, label='VIP
Threshold (>= 1.0)')

ax1.set_title('A. Discriminant Model (PLS-DA VIP)', fontsize=13.5,
fontweight='bold', pad=10)
ax1.set_ylabel('VIP Score', fontsize=12.5, fontweight='bold')
ax1.set_xticks(x)
ax1.set_xticklabels(traits, rotation=35, ha='right', fontsize=10.5,
fontweight='bold')
ax1.grid(True, linestyle=':', alpha=0.6)

# ----------------------------------------------------------
# Panel B: Continuous PLSR VIP Scores (Unsplit Target Model)
# ----------------------------------------------------------
ax2.bar(x - width / 2, df_metrics_master['PLSR_PC1_VIP'], width,
color='#28598a')
ax2.bar(x + width / 2, df_metrics_master['PLSR_PC2_VIP'], width,
color='#d95f02')
ax2.axhline(1.0, color='red', linestyle='--', linewidth=1.6)

ax2.set_title('B. Continuous Model (PLSR VIP)', fontsize=13.5,
fontweight='bold', pad=10)
ax2.set_xticks(x)
ax2.set_xticklabels(traits, rotation=35, ha='right', fontsize=10.5,
fontweight='bold')
ax2.grid(True, linestyle=':', alpha=0.6)

# Unified Y-axis limit across both panels
```

```
max_vip_val = df_metrics_master[['PC1_VIP', 'PC2_VIP', 'PLSR_PC1_VIP',
'PLSR_PC2_VIP']].max().max()
ax1.set_ylim(0, max(max_vip_val + 0.12, 1.45))

# Single shared legend centered cleanly above both panels
handles, labels = ax1.get_legend_handles_labels()
fig.legend(handles, labels, loc='upper center', bbox_to_anchor=(0.5,
1.03),
           ncol=3, fontsize=11, frameon=True)

fig.suptitle('Comparison of Variable Importance in Projection (VIP): PLS-
DA vs. Continuous PLSR',
             fontsize=15, fontweight='bold', y=1.09)

plt.tight_layout()
plt.savefig('coupled_vip_plsda_vs_plsr.png', dpi=300, bbox_inches='tight')
plt.show()

#
================================================================================
====
# 3. FIGURE 1: 2x5 GRID OF ROC CURVES FOR ALL 10 PLANT TRAITS
#
================================================================================
====
fig, axes = plt.subplots(2, 5, figsize=(18, 8), sharex=True, sharey=True)
axes = axes.flatten()

for i, trait in enumerate(plant_traits):
    ax = axes[i]
    sub = df_all_samples[df_all_samples['Trait'] == trait].copy()
    y_bin = (sub['Value'] > sub['Value'].median()).astype(int)
    X_sub = sub[['PC1', 'PC2']]

    pls = PLSRegression(n_components=2)
    pls.fit(X_sub, y_bin)
    scores = pls.predict(X_sub).ravel()
    fpr, tpr, _ = roc_curve(y_bin, scores)
    roc_auc = auc(fpr, tpr)

    ax.plot(fpr, tpr, color='#1f77b4', lw=2.5, label=f'AUC =
{roc_auc:.2f}')
    ax.plot([0, 1], [0, 1], color='gray', linestyle='--', lw=1)
    ax.set_title(f'{trait}', fontsize=12, fontweight='bold')
    ax.legend(loc='lower right', fontsize=10)
```

```
    ax.grid(True, linestyle=':', alpha=0.6)

fig.supxlabel('False Positive Rate (1 - Specificity)', fontsize=12,
fontweight='bold')
fig.supylabel('True Positive Rate (Sensitivity)', fontsize=12,
fontweight='bold')
fig.suptitle('Individual PLSR Discriminant ROC Curves for All 10 Plant
Traits', fontsize=16, fontweight='bold', y=1.02)
plt.tight_layout()
plt.savefig('all_10_roc_grid.png', dpi=300, bbox_inches='tight')
plt.show()

#
================================================================================
====
# 4. FIGURE : COUPLED MODEL EVALUATION (AUC VS. VIP SCORE)
#
================================================================================
====

fig, ax = plt.subplots(figsize=(10, 7))

# 1. Draw connecting lines (dumbbells) between PC1_VIP and PC2_VIP for
each trait
for _, row in df_metrics_AUC.iterrows():
    ax.plot(
        [row['PC1_VIP'], row['PC2_VIP']],
        [row['AUC'], row['AUC']],
        color='gray',
        linestyle='-',
        linewidth=1.5,
        alpha=0.6,
        zorder=2
    )

# 2. Plot PC1 (Thermal) VIP points
ax.scatter(
    df_metrics_AUC['PC1_VIP'],
    df_metrics_AUC['AUC'],
    color='#2b5c8f',
    marker='o',
    s=180,
    label='PC1 (Thermal)',
    zorder=3
)
```

```
# 3. Plot PC2 (Hydro-Climatic) VIP points
ax.scatter(
    df_metrics_AUC['PC2_VIP'],
    df_metrics_AUC['AUC'],
    color='#d95f02',
    marker='s',
    s=180,
    label='PC2 (Hydro-Climatic)',
    zorder=3
)

# 4. Annotate trait names next to the rightmost point (Max_VIP) to avoid
overlapping the line
df_temp = df_metrics_AUC.copy()
df_temp['right_x'] = df_temp[['PC1_VIP', 'PC2_VIP']].max(axis=1).round(4)
df_temp['y_round'] = df_temp['AUC'].round(4)

offsets = [(10, 6), (10, -12), (10, 18), (10, -24)]
for (x_val, y_val), group in df_temp.groupby(['right_x', 'y_round']):
    traits_at_point = group['Trait'].tolist()
    for idx, trait_name in enumerate(traits_at_point):
        dx, dy = offsets[idx % len(offsets)]
        ax.annotate(
            trait_name,
            (x_val, y_val),
            xytext=(dx, dy),
            textcoords='offset points',
            ha='left',
            va='center',
            fontsize=10.5,
            fontweight='bold',
            arrowprops=dict(arrowstyle='-', color='gray', lw=0.8) if
len(traits_at_point) > 1 else None
        )

# 5. Threshold lines and styling
ax.axhline(0.75, color='gray', linestyle='--', linewidth=1.2, label='High
AUC Threshold (>= 0.75)')
ax.axvline(1.10, color='gray', linestyle=':', linewidth=1.2, label='High
VIP Threshold (>= 1.10)')

ax.margins(x=0.18, y=0.14)
ax.legend(loc='lower left', fontsize=10, frameon=True)
```

```
ax.set_xlabel('Environmental Factor Importance (VIP Score)', fontsize=12,
fontweight='bold')
ax.set_ylabel('Model Predictive Accuracy (ROC AUC)', fontsize=12,
fontweight='bold')
ax.set_title('Coupled Model Evaluation: Predictive Power (AUC) vs. PC1 &
PC2 Importance (VIP)', fontsize=13.5, fontweight='bold')
ax.grid(True, linestyle='--', alpha=0.5)

plt.tight_layout()
plt.savefig('coupled_performance_importance_dumbbell.png', dpi=300)
plt.show()


###############AUC vs. R2Y
# Scatter plot coupling Continuous PLSR R2Y (x) with PLS-DA ROC AUC (y)
sns.scatterplot(
    data=df_metrics_master,
    x='PLSR_R2Y',
    y='AUC',
    hue='Dominant_Index',
    style='Dominant_Index',
    palette={'PC1 (Thermal)': '#2b5c8f', 'PC2 (Hydro-Climatic)':
'#d95f02'},
    s=220,
    zorder=3,
    ax=ax
)

# Stagger labels for traits that share nearly identical (PLSR_R2Y, AUC)
coordinates
df_temp = df_metrics_master.copy()
df_temp['x_round'] = df_temp['PLSR_R2Y'].round(3)
df_temp['y_round'] = df_temp['AUC'].round(3)

offsets = [(9, 6), (9, -12), (-9, 6), (-9, -12)]
for (x_val, y_val), group in df_temp.groupby(['x_round', 'y_round']):
    traits_at_point = group['Trait'].tolist()
    for idx, trait_name in enumerate(traits_at_point):
        dx, dy = offsets[idx % len(offsets)]
        ha = 'left' if dx > 0 else 'right'
        ax.annotate(
            trait_name,
            (x_val, y_val),
            xytext=(dx, dy),
            textcoords='offset points',
```

```
            ha=ha,
            va='center',
            fontsize=10.5,
            fontweight='bold',
            arrowprops=dict(arrowstyle='-', color='gray', lw=0.8) if
len(traits_at_point) > 1 else None
        )

# Reference threshold lines defining the four response quadrants
ax.axhline(0.75, color='gray', linestyle='--', linewidth=1.2, label='High
AUC Threshold (>= 0.75)')
ax.axvline(0.50, color='gray', linestyle=':', linewidth=1.2,
label='Moderate-Strong R²Y Threshold (>= 0.50)')

# Optional subtle quadrant annotations in the corners
ax.text(0.98, 0.97, 'Linear & Threshold\nClimate Responders',
transform=ax.transAxes,
        ha='right', va='top', fontsize=9.5, style='italic',
color='#333333', alpha=0.8)
ax.text(0.02, 0.97, 'Non-Linear / Step-Change\nClimate Responders',
transform=ax.transAxes,
        ha='left', va='top', fontsize=9.5, style='italic',
color='#333333', alpha=0.8)

# Place legend cleanly above the plot frame so it never overlaps points
ax.margins(x=0.16, y=0.14)
ax.legend(
    loc='lower center',
    bbox_to_anchor=(0.5, 1.02),
    ncol=2,
    fontsize=10,
    frameon=True
)

ax.set_xlabel('Continuous Model Explanatory Power (PLSR R²Y)',
fontsize=12, fontweight='bold')
ax.set_ylabel('Discriminant Model Accuracy (PLS-DA ROC AUC)', fontsize=12,
fontweight='bold')
ax.set_title(
    'Coupled Model Evaluation: Discriminant Power (AUC) vs. Continuous Fit
(R²Y)',
    fontsize=13.5,
    fontweight='bold',
    pad=52
)
```

```
ax.grid(True, linestyle='--', alpha=0.5)

plt.tight_layout()
plt.savefig('coupled_auc_r2y_performance.png', dpi=300,
bbox_inches='tight')
plt.show()
```

**STEP 9: Model cross validation**

```
#
================================================================================
====
# 4B. COMPUTE PER-ITERATION FOLD Q2 & MERGE INTO df_cv_folds
#
================================================================================
====
fold_q2_records = []
for trait in plant_traits:
    sub = df_all_samples[df_all_samples['Trait'] ==
trait].reset_index(drop=True)
    y_all_raw = sub[['Value']].values
    groups_all = sub['Year'].values

    for tr_idx, val_idx in logo.split(y_all_raw, y_all_raw, groups_all):
        train_yrs_fold = sorted(np.unique(groups_all[tr_idx]).tolist())
        val_yr_fold = int(groups_all[val_idx][0])

        scaler_clim_fold = RobustScaler()
        X_clim_tr_scaled =
scaler_clim_fold.fit_transform(df_clim_summary.loc[train_yrs_fold,
climate_features])
        pca_fold = PCA(n_components=2)
        pcs_tr_clim = pca_fold.fit_transform(X_clim_tr_scaled)

        X_clim_val_scaled =
scaler_clim_fold.transform(df_clim_summary.loc[[val_yr_fold],
climate_features])
        pcs_val_clim = pca_fold.transform(X_clim_val_scaled)

        df_pcs_fold = pd.DataFrame(
            np.vstack([pcs_tr_clim, pcs_val_clim]),
            index=train_yrs_fold + [val_yr_fold],
            columns=pc_cols
        )
        X_f_tr = df_pcs_fold.loc[groups_all[tr_idx], pc_cols].values
```

```
        X_f_val = df_pcs_fold.loc[groups_all[val_idx], pc_cols].values

        y_f_tr_raw = y_all_raw[tr_idx]
        y_f_val_raw = y_all_raw[val_idx].ravel()

        scaler_y_fold = RobustScaler()
        y_f_tr_scaled = scaler_y_fold.fit_transform(y_f_tr_raw).ravel()

        plsr_fold = PLSRegression(n_components=2, scale=False)
        plsr_fold.fit(X_f_tr, y_f_tr_scaled)
        val_pred_orig =
scaler_y_fold.inverse_transform(plsr_fold.predict(X_f_val).reshape(-1,
1)).ravel()

        # Fold-level Q2 = 1 - (PRESS_fold / TSS_fold)
        press_f = np.sum((y_f_val_raw - val_pred_orig) ** 2)
        tss_f = np.sum((y_f_val_raw - np.mean(y_f_val_raw)) ** 2)
        fold_q2 = float(1.0 - (press_f / tss_f)) if tss_f > 0 else np.nan

        fold_q2_records.append({
            'Trait': trait,
            'Held_Out_Year': val_yr_fold,
            'Fold_Q2': round(fold_q2, 2)
        })

df_fold_q2 = pd.DataFrame(fold_q2_records)
if 'Fold_Q2' in cv_fold_details_list.columns:
    cv_fold_details_list = cv_fold_details_list.drop(columns=['Fold_Q2'])
cv_fold_details_list = cv_fold_details_list.merge(df_fold_q2, on=['Trait',
'Held_Out_Year'], how='left')

#
================================================================================
====
# 5. MAIN CV FIGURE (2x2): SUMMARY & PER-ITERATION Q2 (TOP) + PLS-DA
(BOTTOM)
#
================================================================================
====
traits_order = cv_summary_list['Trait'].tolist()
x = np.arange(len(traits_order))
w = 0.38

color_insample = '#2b5c8f'
color_cv = '#d95f02'
```

```
fig, axes = plt.subplots(2, 2, figsize=(17.5, 12.5))

# ------------------------------------------------------------------------
------
# Panel A: Continuous PLSR Summary (In-Sample R²Y vs. 4-Fold Out-of-Sample
Q²)
# ------------------------------------------------------------------------
------
axes[0, 0].bar(x - w/2, cv_summary_list['PLSR_R2Y_FullFit'], width=w,
               label='$R^2Y$ (In-Sample Fit)', color=color_insample)
axes[0, 0].bar(x + w/2, cv_summary_list['PLSR_Q2_4Fold_CV'], width=w,
               label='$Q^2$ (Overall 4-Fold LOOCV)', color=color_cv)
axes[0, 0].axhline(0.50, color='#2ca02c', linestyle='--', linewidth=1.3,
                   label='Strong Predictive Threshold ($Q^2 \\geq 0.50$)')
axes[0, 0].axhline(0.0, color='black', linestyle='-', linewidth=1.0)
axes[0, 0].set_xticks(x)
axes[0, 0].set_xticklabels(traits_order, rotation=30, ha='right',
fontweight='bold')
axes[0, 0].set_ylabel('Explained / Predicted Variance ($R^2Y$ & $Q^2$)',
fontsize=11, fontweight='bold')
axes[0, 0].set_title('A. Continuous PLSR Summary: Model Fit ($R^2Y$) vs.
4-Fold $Q^2$',
                     fontsize=12.5, fontweight='bold', pad=10)
axes[0, 0].legend(loc='upper right', fontsize=9.5, frameon=True)
axes[0, 0].grid(True, linestyle=':', alpha=0.6)

# ------------------------------------------------------------------------
------
# Panel B: Continuous PLSR Iteration-Level Q² Heatmap (Aligned Trait
Order)
# ------------------------------------------------------------------------
------
pivot_q2 = cv_fold_details_list.pivot(index='Held_Out_Year',
columns='Trait', values='Fold_Q2')[traits_order]
sns.heatmap(
    pivot_q2,
    annot=True,
    fmt='.2f',
    cmap='RdYlGn',
    vmin=-1.0,
    vmax=1.0,
    center=0.0,
    cbar_kws={'label': 'Iteration $Q^2$'},
    annot_kws={'size': 9.5, 'fontweight': 'bold'},
```

```
    linewidths=0.6,
    linecolor='white',
    ax=axes[0, 1]
)
axes[0, 1].set_title('B. Continuous PLSR Iteration-Level $Q^2$ for Each
Held-Out Year',
                     fontsize=12.5, fontweight='bold', pad=10)
axes[0, 1].set_xlabel('')
axes[0, 1].set_ylabel('Held-Out Validation Year (Iteration)', fontsize=11,
fontweight='bold')
axes[0, 1].set_xticklabels(axes[0, 1].get_xticklabels(), rotation=30,
ha='right', fontweight='bold')
axes[0, 1].set_yticklabels(axes[0, 1].get_yticklabels(), rotation=0,
fontweight='bold')

# ------------------------------------------------------------------------
------
# Panel C: Discriminant PLS-DA Summary (In-Sample AUC vs. 4-Fold CV AUC)
# ------------------------------------------------------------------------
------
axes[1, 0].bar(x - w/2, cv_summary_list['PLSDA_AUC_FullFit'], width=w,
               label='In-Sample Discrimination (Full AUC)',
color=color_insample)
axes[1, 0].bar(x + w/2, cv_summary_list['PLSDA_AUC_4Fold_CV'], width=w,
               label='4-Fold Out-of-Sample (CV AUC)', color=color_cv)
axes[1, 0].axhline(0.75, color='#2ca02c', linestyle='--', linewidth=1.3,
                   label='Strong Classification Threshold (AUC ≥ 0.75)')
axes[1, 0].axhline(0.50, color='gray', linestyle=':', linewidth=1.1,
                   label='Random Classifier Baseline (0.50)')
axes[1, 0].set_ylim(0.35, 1.05)
axes[1, 0].set_xticks(x)
axes[1, 0].set_xticklabels(traits_order, rotation=30, ha='right',
fontweight='bold')
axes[1, 0].set_ylabel('Discriminant Accuracy (ROC AUC)', fontsize=11,
fontweight='bold')
axes[1, 0].set_title('C. Discriminant PLS-DA Summary: Full AUC vs. 4-Fold
CV AUC',
                     fontsize=12.5, fontweight='bold', pad=10)
axes[1, 0].legend(loc='lower right', fontsize=9.5, frameon=True)
axes[1, 0].grid(True, linestyle=':', alpha=0.6)

# ------------------------------------------------------------------------
------
# Panel D: Discriminant PLS-DA Iteration-Level Accuracy Heatmap (No Blank
Cells)
```

```
# ------------------------------------------------------------------------
------
pivot_acc = cv_fold_details_list.pivot(index='Held_Out_Year',
columns='Trait', values='Fold_PLSDA_Acc')[traits_order]
sns.heatmap(
    pivot_acc,
    annot=True,
    fmt='.2f',
    cmap='Blues',
    vmin=0.0,
    vmax=1.0,
    cbar_kws={'label': 'Iteration PLS-DA Accuracy (0.00-1.00)'},
    annot_kws={'size': 10, 'fontweight': 'bold'},
    linewidths=0.6,
    linecolor='white',
    ax=axes[1, 1]
)
axes[1, 1].set_title('D. Discriminant PLS-DA Iteration-Level Accuracy for
Each Held-Out Year',
                     fontsize=12.5, fontweight='bold', pad=10)
axes[1, 1].set_xlabel('Plant Trait', fontsize=11, fontweight='bold')
axes[1, 1].set_ylabel('Held-Out Validation Year (Iteration)', fontsize=11,
fontweight='bold')
axes[1, 1].set_xticklabels(axes[1, 1].get_xticklabels(), rotation=30,
ha='right', fontweight='bold')
axes[1, 1].set_yticklabels(axes[1, 1].get_yticklabels(), rotation=0,
fontweight='bold')

plt.suptitle(
    'Coupled 4-Fold Leave-One-Year-Out Cross-Validation: Summary vs.
Iteration-Level Stability\n'
    '(PC1 Air/Soil Temp & Rain [65.53%] | PC2 Wind & Soil Moist
[21.55%])',
    fontsize=14.5, fontweight='bold', y=1.02
)
plt.tight_layout()
plt.savefig('/content/drive/MyDrive/Coupled_PLSR_PLSDA_Q2_AUC_Acc_Summary_
and_Iterations.png', dpi=300, bbox_inches='tight')
plt.show()
```