Combining all the data

```
# collecting Y prediction datasets
import os
import gzip
import pandas as pd

# Define folder path where the files are stored
folder_path = "/Users/etloaner/Documents/ASU/Capstone project/Storm event dataset"

# Function to read compressed CSV files
def read_gz_csv(file_path):
    with gzip.open(file_path, "rt") as f:
        return pd.read_csv(f, low_memory=False)

# Lists to store data
details_list = []
locations_list = []
fatalities_list = []

# Read each file and append to corresponding list
for file in os.listdir(folder_path):
    file_path = os.path.join(folder_path, file)

    if "details" in file:
        df = read_gz_csv(file_path)
        df.columns = df.columns.str.strip().str.lower()  # Standardizing column names
        details_list.append(df)
    elif "locations" in file:
        df = read_gz_csv(file_path)
        df.columns = df.columns.str.strip().str.lower()
        locations_list.append(df)
    elif "fatalities" in file:
        df = read_gz_csv(file_path)
        df.columns = df.columns.str.strip().str.lower()
        fatalities_list.append(df)

# Concatenate all dataframes (append)
details_df = pd.concat(details_list, ignore_index=True) if details_list else pd.DataFrame()
locations_df = pd.concat(locations_list, ignore_index=True) if locations_list else pd.DataFran
fatalities_df = pd.concat(fatalities_list, ignore_index=True) if fatalities_list else pd.DataF

# Print column names for debugging
print("Details Columns:", details_df.columns)
print("Locations Columns:", locations_df.columns)
print("Fatalities Columns:", fatalities_df.columns)

# Ensure 'event_id' exists in all DataFrames before merging
if "event_id" not in details_df.columns:
    raise KeyError("event_id missing from details dataset")
if "event_id" not in locations_df.columns:
    raise KeyError("event_id missing from locations dataset")
if "event_id" not in fatalities_df.columns:
```

```
    raise KeyError("event_id missing from fatalities dataset")

# Merge data using event_id
merged_df = details_df.merge(locations_df, on="event_id", how="left") \
                      .merge(fatalities_df, on="event_id", how="left")

# Save final dataframe to CSV
output_file = "final_storm_data.csv"
merged_df.to_csv(output_file, index=False)

print(f"Final consolidated dataset saved as {output_file}")
```

Cleaning and filtering the data

```
import pandas as pd
storm_df = pd.read_csv("/Users/etloaner/Documents/ASU/Capstone project/final_storm_data.csv")


import pandas as pd
import numpy as np

# Function to convert damage strings (e.g., "10.00K", "10.00M") to a numeric value
def convert_damage(damage_str):
    try:
        if pd.isnull(damage_str):
            return 0.0
        damage_str = str(damage_str).strip()
        if damage_str.endswith('K'):
            return float(damage_str[:-1]) * 1e3
        elif damage_str.endswith('M'):
            return float(damage_str[:-1]) * 1e6
        else:
            return float(damage_str)
    except Exception:
        return 0.0

# Function to categorize an event as extreme (1) or non-extreme (0)
def categorize_extreme(row):
    # Get the event type and ensure no extra spaces
    event_type = str(row.get('event_type', '')).strip()

    # Convert damages to numeric values
    damage_property = convert_damage(row.get('damage_property', '0.00K'))
    damage_crops = convert_damage(row.get('damage_crops', '0.00K'))
    total_damage = damage_property + damage_crops

    # Sum injuries and deaths from direct and indirect counts
    total_injuries = (row.get('injuries_direct', 0) or 0) + (row.get('injuries_indirect', 0)
    total_deaths = (row.get('deaths_direct', 0) or 0) + (row.get('deaths_indirect', 0) or 0)

    # Set default thresholds (adjust these based on your domain knowledge)
    damage_threshold = 100000       # Example: $100K damage threshold
    injury_threshold = 20           # Example: 20 or more injuries
    death_threshold = 5             # Example: 5 or more deaths
```

```
    wind_speed_threshold = 50       # Wind speed in knots for extreme wind events
    hail_size_threshold = 1.5       # Hail size in inches considered extreme

    # Overall impact: if damage, injuries, or deaths exceed thresholds, mark as extreme.
    if total_damage >= damage_threshold or total_injuries >= injury_threshold or total_deaths
        return 1

    # Specific rules per event type
    if event_type == 'Tornado':
        # Use the Enhanced Fujita Scale: EF2 or higher is considered extreme.
        tor_scale = str(row.get('tor_f_scale', 'EF0')).strip()
        try:
            scale_value = int(tor_scale.replace('EF', ''))
        except Exception:
            scale_value = 0
        if scale_value >= 2:
            return 1

    if event_type in ['Thunderstorm Wind', 'High Wind', 'Strong Wind', 'Marine Thunderstorm Wi
        # For wind events, use the magnitude (assumed to be in knots)
        try:
            magnitude = float(row.get('magnitude', 0))
        except Exception:
            magnitude = 0
        if magnitude >= wind_speed_threshold:
            return 1

    if event_type == 'Hail':
        # For hail events, use the magnitude (assumed to be in inches)
        try:
            magnitude = float(row.get('magnitude', 0))
        except Exception:
            magnitude = 0
        if magnitude >= hail_size_threshold:
            return 1

    if event_type in ['Flash Flood', 'Flood', 'Coastal Flood']:
        # Flood events may be extreme if they cause high damage (already checked above)
        if total_damage >= damage_threshold:
            return 1

    if event_type in ['Heavy Rain', 'Excessive Heat', 'Heat']:
        # These events may be extreme if they result in significant injuries or damage.
        if total_injuries >= injury_threshold or total_damage >= damage_threshold:
            return 1

    if event_type in ['Winter Storm', 'Winter Weather', 'Blizzard']:
        # Winter events are flagged based on impact
        if total_damage >= damage_threshold or total_injuries >= injury_threshold:
            return 1

    # Additional event-specific rules can be added here

    # If none of the conditions are met, mark as non-extreme.
    return 0
```

```
# Load your consolidated dataset (ensure that column names are standardized, e.g., lower-case
df = pd.read_csv("final_storm_data.csv")
df.columns = df.columns.str.strip().str.lower()  # e.g., converting "Event_Type" to "event_typ

# Apply the categorization function to each row in the DataFrame
df['extreme'] = df.apply(categorize_extreme, axis=1)

# Save the updated DataFrame with the extreme flag to a new CSV file
output_file = "final_storm_data_with_extreme_flag.csv"
df.to_csv(output_file, index=False)

print(f"Categorization complete. Saved final file as '{output_file}'.")
```

Getting only last few years data

```
import pandas as pd
final_extreme_event_dataset = pd.read_csv("/Users/etloaner/Documents/ASU/Capstone project/fina
filtered_df = final_extreme_event_dataset[final_extreme_event_dataset["year"].isin([2012, 2013
filtered_df.drop(columns=["episode_id_x","event_id","state_fips","cz_type","cz_fips","cz_name"
filtered_df = filtered_df[["begin_day", "begin_time", "end_day", "end_time", "event_type", "be
filtered_df.dropna(inplace=True)
filtered_df.drop(columns=["begin_day","begin_time","end_day","end_time"], inplace=True)

import pandas as pd
from datetime import datetime
import pytz

df = filtered_df.copy()
# Mapping of timezone abbreviations to UTC offsets
timezone_offsets = {
    'CST-6': -6,
    'EST-5': -5,
    'MST-7': -7,
    'PST-8': -8,
    'AST-4': -4,
    'HST-10': -10,
    'AKST-9': -9,
    'SST-11': -11,
    'GST10': 10
}

# Function to convert local time to UTC
def convert_to_utc(local_time_str, timezone):
    local_time = datetime.strptime(local_time_str, '%d-%b-%y %H:%M:%S')
    offset = timezone_offsets.get(timezone, 0)
    local_time = local_time.replace(tzinfo=pytz.FixedOffset(offset * 60))
    utc_time = local_time.astimezone(pytz.utc)
    return utc_time

# Apply the function to the begin_date_time and end_date_time columns
df['begin_date_time_utc'] = df.apply(lambda row: convert_to_utc(row['begin_date_time'], row['c
df['end_date_time_utc'] = df.apply(lambda row: convert_to_utc(row['end_date_time'], row['cz_ti

# Split the datetime objects into separate date and time columns
```

```
df['begin_date_utc'] = df['begin_date_time_utc'].dt.date
df['begin_time_utc'] = df['begin_date_time_utc'].dt.time
df['end_date_utc'] = df['end_date_time_utc'].dt.date
df['end_time_utc'] = df['end_date_time_utc'].dt.time

# Drop the intermediate UTC datetime columns
df.drop(columns=['begin_date_time_utc', 'end_date_time_utc'], inplace=True)
```

```
df.to_csv("final_storm_data_with_utc.csv", index=False)
```

```
df.head()
```

```
import pandas as pd
df = pd.read_csv("final_storm_data_with_utc.csv")

# Convert the time columns to datetime objects
df['begin_date_time'] = pd.to_datetime(df['begin_date_time'], format='%d-%b-%y %H:%M:%S')
df['end_date_time'] = pd.to_datetime(df['end_date_time'], format='%d-%b-%y %H:%M:%S')

# Filter the DataFrame for events that lasted for some time (i.e., end time greater than start
filtered_df = df[df['end_date_time'] > df['begin_date_time']]

# Count the rows where the start and end times are exactly the same (i.e., instantaneous event
same_time_count = df[df['begin_date_time'] == df['end_date_time']].shape[0]

# Display results
print("Filtered events (lasting more than 0 seconds):")
print(filtered_df)
print("\nNumber of rows with identical start and end times on the same day:", same_time_count)
```

```
filtered_df = df[df['end_date_time'] > df['begin_date_time']]
same_time_count = df[df['begin_date_time'] == df['end_date_time']].shape[0]
print(filtered_df.shape[0])
```

```
import pandas as pd

# Convert the columns to datetime objects
df['begin_date_time'] = pd.to_datetime(df['begin_date_time'], format='%d-%b-%y %H:%M:%S')
df['end_date_time'] = pd.to_datetime(df['end_date_time'], format='%d-%b-%y %H:%M:%S')

# Define a 20 minutes threshold
time_threshold = pd.Timedelta(minutes=120)

# Filter the DataFrame for events with a duration longer than the threshold
filtered_df = df[(df['end_date_time'] - df['begin_date_time']) > time_threshold]

# Count the rows where the start and end times are exactly the same (instantaneous events)
same_time_count = df[df['begin_date_time'] == df['end_date_time']].shape[0]

# Display the filtered DataFrame and the count of instantaneous events
print("Filtered events (lasting more than 20 minutes):")
print(filtered_df)
print("\nNumber of rows with identical start and end times on the same day:", same_time_count)
```

```python
filtered_df.head()
```

```python
filtered_df.drop_duplicates(inplace=True)
```

```python
# Define grid size based on satellite resolution
import pandas as pd
import numpy as np
GRID_SIZE = 0.02  # approximately 2km in decimal degrees

# Round coordinates to match satellite data resolution
filtered_df['begin_lat'] = np.round(filtered_df['begin_lat'] / GRID_SIZE) * GRID_SIZE
filtered_df['begin_lon'] = np.round(filtered_df['begin_lon'] / GRID_SIZE) * GRID_SIZE
filtered_df['end_lat'] = np.round(filtered_df['end_lat'] / GRID_SIZE) * GRID_SIZE
filtered_df['end_lon'] = np.round(filtered_df['end_lon'] / GRID_SIZE) * GRID_SIZE

# Remove duplicates based on gridded coordinates
filtered_df = filtered_df.drop_duplicates(subset=[
    'event_type',
    'begin_date_time',
    'begin_lat',
    'begin_lon',
    'end_lat',
    'end_lon'
])
```

```python
from sklearn.cluster import DBSCAN
import numpy as np
from math import radians, cos, sin, asin, sqrt

def haversine_distance(lat1, lon1, lat2, lon2):
    """Calculate distance between two points in kilometers"""
    R = 6371  # Earth's radius in kilometers

    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1

    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    return R * c

# Create coordinate array
coords = filtered_df[['begin_lat', 'begin_lon']].values

# DBSCAN with eps=0.018 (approximately 2km)
# eps is in degrees, 0.018 degrees ≈ 2km at the equator
clustering = DBSCAN(eps=0.018, min_samples=1).fit(coords)

# Add cluster labels to DataFrame
filtered_df['cluster'] = clustering.labels_

# Keep one event per cluster
filtered_df = filtered_df.groupby(['cluster', 'event_type', 'begin_date_time']).first().reset_
filtered_df.drop('cluster', axis=1, inplace=True)
```

```python
filtered_df.head()
```

```python
filtered_df.info()
```

```python
from sklearn.cluster import KMeans
import folium
import numpy as np
from folium.plugins import MarkerCluster

# Extract coordinates for clustering
coordinates = filtered_df[['begin_lat', 'begin_lon']].values

# Perform KMeans clustering
kmeans = KMeans(n_clusters=50, random_state=42)
filtered_df['cluster'] = kmeans.fit_predict(coordinates)

# Get cluster centers
cluster_centers = kmeans.cluster_centers_

# Create a base map centered on the mean coordinates
center_lat = filtered_df['begin_lat'].mean()
center_lon = filtered_df['begin_lon'].mean()
m = folium.Map(location=[center_lat, center_lon], zoom_start=4)

# Add a MarkerCluster layer for all points
marker_cluster = MarkerCluster().add_to(m)

# Plot all points with cluster colors
colors = [f'#{hash(str(c)) % 0xFFFFFF:06x}' for c in range(50)]  # Generate unique colors

# Add individual points with cluster colors
for idx, row in filtered_df.iterrows():
    folium.CircleMarker(
        location=[row['begin_lat'], row['begin_lon']],
        radius=3,
        color=colors[row['cluster']],
        fill=True,
        popup=f"Cluster: {row['cluster']}<br>Event: {row['event_type']}"
    ).add_to(marker_cluster)

# Add cluster centers with larger markers
for i, center in enumerate(cluster_centers):
    folium.CircleMarker(
        location=[center[0], center[1]],
        radius=8,
        color='red',
        fill=True,
        popup=f'Cluster Center {i}'
    ).add_to(m)

# Save the map
m.save('weather_clusters.html')

# Print cluster statistics
print("\nCluster Statistics:")
```

```python
cluster_stats = filtered_df.groupby('cluster').agg({
    'event_type': 'count',
    'begin_lat': ['mean', 'std'],
    'begin_lon': ['mean', 'std']
}).round(4)
print(cluster_stats)
```

```python
filtered_df["extreme"].value_counts()
```

```python
import pandas as pd
from datetime import datetime, timedelta

def create_final_dataset(filtered_df, api_weather_data):
    """
    Merge extreme events data with weather API data
    """
    # Convert datetime columns to consistent format
    filtered_df['date'] = pd.to_datetime(filtered_df['begin_date_time'])
    api_weather_data['date'] = pd.to_datetime(api_weather_data['date'])

    # Round coordinates to 4 decimal places for matching
    filtered_df['lat_rounded'] = filtered_df['begin_lat'].round(4)
    filtered_df['lon_rounded'] = filtered_df['begin_lon'].round(4)
    api_weather_data['lat_rounded'] = api_weather_data['latitude'].round(4)
    api_weather_data['lon_rounded'] = api_weather_data['longitude'].round(4)

    # Merge datasets based on date and location
    merged_df = pd.merge(
        filtered_df,
        api_weather_data,
        how='left',
        left_on=['date', 'lat_rounded', 'lon_rounded'],
        right_on=['date', 'lat_rounded', 'lon_rounded']
    )

    # Clean up merged dataset
    # Drop duplicate columns and temporary columns
    columns_to_drop = ['lat_rounded', 'lon_rounded', 'city', 'latitude', 'longitude']
    merged_df = merged_df.drop(columns=columns_to_drop)

    # Check for missing values
    missing_data = merged_df.isnull().sum()
    print("\nMissing values in merged dataset:")
    print(missing_data[missing_data > 0])

    # Save final dataset
    merged_df.to_csv('final_weather_events_dataset.csv', index=False)
    print("\nFinal dataset shape:", merged_df.shape)

    return merged_df

# Load the API weather data
api_weather_data = pd.read_csv('us_weather_data.csv')

# Create final dataset
```

```python
final_df = create_final_dataset(filtered_df, api_weather_data)

# Verify the merge
print("\nSample of final dataset:")
print(final_df.head())

# Check class distribution in final dataset
print("\nExtreme event distribution in final dataset:")
print(final_df['extreme'].value_counts(normalize=True))
```

```python
filtered_df.head()
```

```python
def get_unique_locations(filtered_df):
    """Extract unique locations and dates from events dataframe"""
    locations = filtered_df.groupby(['begin_lat', 'begin_lon']).agg({
        'begin_date_time': ['min', 'max']
    }).reset_index()

    locations.columns = ['lat', 'lon', 'start_date', 'end_date']
    locations['start_date'] = pd.to_datetime(locations['start_date']).dt.strftime('%Y-%m-%d')
    locations['end_date'] = pd.to_datetime(locations['end_date']).dt.strftime('%Y-%m-%d')

    return locations.to_dict('records')

# Get unique locations from the filtered dataset
unique_locations = get_unique_locations(filtered_df)
```

```python
len(unique_locations)
```

```python
import openmeteo_requests
import requests_cache
import pandas as pd
from retry_requests import retry
import time
from tqdm import tqdm
from datetime import datetime, timedelta

def get_unique_locations_with_72h(filtered_df):
    """
    Extract unique locations and date ranges (previous 72 hours) from events dataframe.
    """
    # Add a column for 72 hours prior to each event's start time
    filtered_df['start_date_72h'] = pd.to_datetime(filtered_df['begin_date_time']) - timedelta
    filtered_df['start_date_72h'] = filtered_df['start_date_72h'].dt.strftime('%Y-%m-%d')
    filtered_df['end_date'] = pd.to_datetime(filtered_df['begin_date_time']).dt.strftime('%Y-%

    # Group by locations and date ranges to minimize API calls
    locations = filtered_df.groupby(['begin_lat', 'begin_lon', 'start_date_72h', 'end_date']).
    locations.columns = ['lat', 'lon', 'start_date', 'end_date', 'count']

    return locations.to_dict('records')

def fetch_weather_data_optimized(filtered_df, batch_size=10):
    """
    Fetch weather data for all unique locations with batching.
```

```
    This function minimizes API calls by grouping requests by location and date range.
    """
    # Setup the Open-Meteo API client with caching and retries
    cache_session = requests_cache.CachedSession('.cache', expire_after=-1)
    retry_session = retry(cache_session, retries=5, backoff_factor=0.2)
    openmeteo = openmeteo_requests.Client(session=retry_session)

    # Get unique locations with 72-hour ranges
    locations = get_unique_locations_with_72h(filtered_df)
    print(f"Total unique location-date combinations: {len(locations)}")

    # Initialize list for all weather data
    all_weather_data = []

    # Process locations in batches
    for i in tqdm(range(0, len(locations), batch_size), desc="Fetching weather data"):
        batch = locations[i:i + batch_size]

        for loc in batch:
            params = {
                "latitude": loc["lat"],
                "longitude": loc["lon"],
                "start_date": loc["start_date"],
                "end_date": loc["end_date"],
                "hourly": [
                    "temperature_2m",
                    "relative_humidity_2m",
                    "dew_point_2m",
                    "apparent_temperature",
                    "precipitation",
                    "rain",
                    "snowfall",
                    "snow_depth",
                    "weather_code",
                    "pressure_msl",
                    "surface_pressure",
                    "cloud_cover",
                    "wind_speed_10m",
                    "wind_direction_10m",
                    "wind_gusts_10m",
                    "soil_temperature_0_to_7cm",
                    "soil_moisture_0_to_7cm"
                ]
            }

            try:
                responses = openmeteo.weather_api(
                    "https://archive-api.open-meteo.com/v1/archive",
                    params=params
                )
                response = responses[0]

                # Process hourly data
                hourly = response.Hourly()
                data = {
                    "date": pd.date_range(
```

```
                        start=pd.to_datetime(hourly.Time(), unit="s", utc=True),
                        end=pd.to_datetime(hourly.TimeEnd(), unit="s", utc=True),
                        freq=pd.Timedelta(seconds=hourly.Interval()),
                        inclusive="left"
                    ),
                    "latitude": loc["lat"],
                    "longitude": loc["lon"]
                }

                # Add all weather variables
                for idx, var in enumerate(params["hourly"]):
                    data[var] = hourly.Variables(idx).ValuesAsNumpy()

                all_weather_data.append(pd.DataFrame(data))

            except Exception as e:
                print(f"\nError fetching data for location {loc['lat']}, {loc['lon']}: {str(e)
                continue

        # Rate limiting to avoid hitting API limits
        time.sleep(1)

    # Combine all data into a single DataFrame
    if all_weather_data:
        weather_df = pd.concat(all_weather_data, ignore_index=True)
        weather_df.to_csv('weather_data.csv', index=False)
        print(f"\nCollected weather data shape: {weather_df.shape}")
        return weather_df
    else:
        raise Exception("No weather data collected")

def create_final_dataset(events_df, weather_df):
    """
    Merge the events dataframe with the collected weather data.
    """
    # Convert timestamps to match for merging
    events_df['datetime'] = pd.to_datetime(events_df['begin_date_time'])

    # Merge on latitude, longitude, and datetime (nearest match within 1 hour)
    final_dataset = pd.merge_asof(
        events_df.sort_values('datetime'),
        weather_df.sort_values('date'),
        left_on='datetime',
        right_on='date',
        by=['begin_lat', 'begin_lon'],
        tolerance=pd.Timedelta('1H'),
        direction='nearest'
    )

    return final_dataset

# Main execution
def main():
    try:
        # Load your events dataframe (replace this with your actual dataframe)
        filtered_df = pd.read_csv("events.csv")  # Replace with your file path
```

```
        # Fetch weather data (optimized to minimize API calls)
        weather_data = fetch_weather_data_optimized(filtered_df)

        # Create final dataset by merging events with weather data
        final_df = create_final_dataset(filtered_df, weather_data)

        print("\nData collection and merging completed successfully!")

        # Save final dataset to a CSV file
        final_df.to_csv("final_dataset.csv", index=False)

        return final_df

    except Exception as e:
        print(f"Error in data collection process: {str(e)}")
        return None

# Run the process
if __name__ == "__main__":
    final_df = main()
```

```
import pandas as pd
from datetime import datetime, timedelta
def get_unique_locations_with_72h(filtered_df):
    """
    Extract unique locations and date ranges (previous 72 hours) from events dataframe.
    """
    # Add a column for 72 hours prior to each event's start time
    filtered_df['start_date_72h'] = pd.to_datetime(filtered_df['begin_date_time']) - timedelta
    filtered_df['start_date_72h'] = filtered_df['start_date_72h'].dt.strftime('%Y-%m-%d')
    filtered_df['end_date'] = pd.to_datetime(filtered_df['begin_date_time']).dt.strftime('%Y-%

    # Group by locations and date ranges to minimize API calls
    locations = filtered_df.groupby(['begin_lat', 'begin_lon', 'start_date_72h', 'end_date']).
    locations.columns = ['lat', 'lon', 'start_date', 'end_date', 'count']

    return locations.to_dict('records')

get_unique_locations_with_72h(filtered_df)
```

```
len(get_unique_locations_with_72h(filtered_df))
```

```
filtered_df.info()
```

```
filtered_df.to_csv("Final_prediction_dataset.csv", index=False)
```

```
filtered_df.head()
```

```
extreme_df = pd.read_csv("Final_prediction_dataset.csv")
extreme_df = extreme_df[:5]
extreme_df.info()
```

```python
import openmeteo_requests
import requests_cache
import pandas as pd
from retry_requests import retry
import time
import pickle
from tqdm import tqdm

# ----------------------------
# Setup API client with caching and retry logic
# ----------------------------
cache_session = requests_cache.CachedSession('.cache', expire_after=-1)
retry_session = retry(cache_session, retries=5, backoff_factor=0.2)
openmeteo = openmeteo_requests.Client(session=retry_session)
url = "https://archive-api.open-meteo.com/v1/archive"

# ----------------------------
# Load the extreme event data
# ----------------------------
extreme_df = pd.read_csv("Final_prediction_dataset.csv")
# Create a UTC datetime column using the provided UTC date and time fields.
extreme_df['event_datetime'] = pd.to_datetime(
    extreme_df['begin_date_utc'] + " " + extreme_df['begin_time_utc'], utc=True
)

# ----------------------------
# Define the hourly weather variables to request.
# ----------------------------
hourly_vars = [
    "temperature_2m", "relative_humidity_2m", "dew_point_2m", "apparent_temperature",
    "precipitation", "rain", "snowfall", "snow_depth", "weather_code", "pressure_msl",
    "surface_pressure", "cloud_cover", "cloud_cover_low", "cloud_cover_mid", "cloud_cover_high
    "et0_fao_evapotranspiration", "vapour_pressure_deficit", "wind_speed_10m", "wind_speed_100
    "wind_direction_10m", "wind_direction_100m", "wind_gusts_10m", "soil_temperature_0_to_7cm"
    "soil_temperature_7_to_28cm", "soil_temperature_28_to_100cm", "soil_temperature_100_to_255
    "soil_moisture_0_to_7cm", "soil_moisture_7_to_28cm", "soil_moisture_28_to_100cm",
    "soil_moisture_100_to_255cm"
]

# ----------------------------
# Process each extreme event (row) individually
# ----------------------------
prev_weather_list = []  # To store each event's 72-hour weather history.
failed_rows = []        # To log any failed API calls.

for idx, row in tqdm(extreme_df.iterrows(), total=len(extreme_df), desc="Processing events"):
    lat = row['begin_lat']
    lon = row['begin_lon']
    event_dt = row['event_datetime']

    # Calculate the start of the 72-hour window
    start_dt = event_dt - pd.Timedelta(hours=72)

    # Format the dates as required by the API (YYYY-MM-DD)
    start_date_str = start_dt.strftime('%Y-%m-%d')
```

```
    end_date_str = event_dt.strftime('%Y-%m-%d')

    # Set API parameters for this event
    params = {
        "latitude": lat,
        "longitude": lon,
        "start_date": start_date_str,
        "end_date": end_date_str,
        "hourly": hourly_vars
    }

    try:
        # Call the API (each call returns a list, so we take the first response)
        responses = openmeteo.weather_api(url, params=params)
        if not responses:
            raise ValueError("Empty response received")
        response = responses[0]
        hourly = response.Hourly()

        # Create a date_range from the hourly data timestamps (all in UTC)
        date_range = pd.date_range(
            start=pd.to_datetime(hourly.Time(), unit="s", utc=True),
            end=pd.to_datetime(hourly.TimeEnd(), unit="s", utc=True),
            freq=pd.Timedelta(seconds=hourly.Interval()),
            inclusive="left"
        )

        # Build a DataFrame from the hourly data
        weather_data = {"date": date_range}
        for i, var in enumerate(hourly_vars):
            weather_data[var] = hourly.Variables(i).ValuesAsNumpy()
        weather_df = pd.DataFrame(weather_data)

        # Filter the DataFrame to only include records within the previous 72 hours
        mask = (weather_df['date'] >= start_dt) & (weather_df['date'] < event_dt)
        history_df = weather_df.loc[mask]
        history_records = history_df.to_dict(orient="records")
        prev_weather_list.append(history_records)

    except Exception as e:
        # Log error details and store None for this event
        failed_rows.append({
            "index": idx,
            "latitude": lat,
            "longitude": lon,
            "start_date": start_date_str,
            "end_date": end_date_str,
            "error": str(e)
        })
        prev_weather_list.append(None)
        # Save partial data immediately in case of error
        with open("prev_weather_partial.pkl", "wb") as f:
            pickle.dump(prev_weather_list, f)
        print(f"Error at row {idx}: {e}. Partial data saved to 'prev_weather_partial.pkl'.")

    # Sleep 0.1 seconds to not exceed 600 calls per minute
```

```
    time.sleep(2)

# ----------------------------
# Save the collected weather history back into the extreme event DataFrame
# ----------------------------
extreme_df['prev_72h_weather'] = prev_weather_list

# Save the final DataFrame (using pickle to preserve the nested structure)
extreme_df.to_pickle("extreme_events_with_prev_weather.pkl")
print("Final data saved to 'extreme_events_with_prev_weather.pkl'.")

# Optionally, save the failed rows details to CSV for further review.
if failed_rows:
    pd.DataFrame(failed_rows).to_csv("failed_rows.csv", index=False)
    print("Some API calls failed. Details saved to 'failed_rows.csv'.")
else:
    print("All API calls succeeded.")
```

```
extreme_df['prev_72h_weather'][0][-2]
```

```
extreme_df
```

```
import os
import openmeteo_requests
import requests_cache
import pandas as pd
from retry_requests import retry
import time
import pickle
from tqdm import tqdm
import logging

# Setup logging to output errors and progress.
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Setup API client with caching and retry logic.
cache_session = requests_cache.CachedSession('.cache', expire_after=-1)
retry_session = retry(cache_session, retries=5, backoff_factor=0.2)
openmeteo = openmeteo_requests.Client(session=retry_session)
url = "https://archive-api.open-meteo.com/v1/archive"

# Load the extreme event data.
extreme_df = pd.read_csv("Final_prediction_dataset.csv")
extreme_df = extreme_df.iloc[:10001,:]
extreme_df['event_datetime'] = pd.to_datetime(
    extreme_df['begin_date_utc'] + " " + extreme_df['begin_time_utc'], utc=True
)

# Define the hourly weather variables.
hourly_vars = [
    "temperature_2m", "relative_humidity_2m", "dew_point_2m", "apparent_temperature",
    "precipitation", "rain", "snowfall", "snow_depth", "weather_code", "pressure_msl",
    "surface_pressure", "cloud_cover", "cloud_cover_low", "cloud_cover_mid", "cloud_cover_high
    "et0_fao_evapotranspiration", "vapour_pressure_deficit", "wind_speed_10m", "wind_speed_10
    "wind_direction_10m", "wind_direction_100m", "wind_gusts_10m", "soil_temperature_0_to_7cm"
```

```
    "soil_temperature_7_to_28cm", "soil_temperature_28_to_100cm", "soil_temperature_100_to_255
    "soil_moisture_0_to_7cm", "soil_moisture_7_to_28cm", "soil_moisture_28_to_100cm",
    "soil_moisture_100_to_255cm"
]

# Attempt to load previous progress if available
if os.path.exists("prev_weather_partial.pkl"):
    with open("prev_weather_partial.pkl", "rb") as f:
        prev_weather_list = pickle.load(f)
    logging.info(f"Resuming from checkpoint. Processed events: {len(prev_weather_list)}.")
else:
    prev_weather_list = []

failed_rows = []

def save_progress(data, filename):
    with open(filename, "wb") as f:
        pickle.dump(data, f)
    logging.info(f"Progress saved to '{filename}'.")

# Determine starting index based on previously processed rows.
start_index = len(prev_weather_list)

# Resume processing from the start_index.
for idx, row in tqdm(extreme_df.iloc[start_index:].iterrows(), total=len(extreme_df) - start_i
    lat = row['begin_lat']
    lon = row['begin_lon']
    event_dt = row['event_datetime']

    start_dt = event_dt - pd.Timedelta(hours=72)
    start_date_str = start_dt.strftime('%Y-%m-%d')
    end_date_str = event_dt.strftime('%Y-%m-%d')

    params = {
        "latitude": lat,
        "longitude": lon,
        "start_date": start_date_str,
        "end_date": end_date_str,
        "hourly": hourly_vars
    }

    try:
        responses = openmeteo.weather_api(url, params=params)
        if not responses:
            raise ValueError("Empty response received")
        response = responses[0]
        print(response)
        break
        hourly = response.Hourly()

        # Create a date_range from the hourly data timestamps.
        date_range = pd.date_range(
            start=pd.to_datetime(hourly.Time(), unit="s", utc=True),
            end=pd.to_datetime(hourly.TimeEnd(), unit="s", utc=True),
            freq=pd.Timedelta(seconds=hourly.Interval()),
            inclusive="left"
```

```
        )

        # Build a DataFrame from the hourly data.
        weather_data = {"date": date_range}
        for i, var in enumerate(hourly_vars):
            weather_data[var] = hourly.Variables(i).ValuesAsNumpy()
        weather_df = pd.DataFrame(weather_data)

        # Filter for records within the previous 72 hours.
        mask = (weather_df['date'] >= start_dt) & (weather_df['date'] < event_dt)
        history_df = weather_df.loc[mask]
        history_records = history_df.to_dict(orient="records")
        prev_weather_list.append(history_records)

    except Exception as e:
        failed_rows.append({
            "index": idx + start_index,  # adjust index to account for skipped rows
            "latitude": lat,
            "longitude": lon,
            "start_date": start_date_str,
            "end_date": end_date_str,
            "error": str(e)
        })
        prev_weather_list.append(None)
        logging.error(f"Error at row {idx + start_index}: {e}. Saving partial data.")
        save_progress(prev_weather_list, "prev_weather_partial.pkl")
        time.sleep(5)
        continue

    # Sleep to control API call frequency.

    time.sleep(2)
    # Optionally save progress every N events.
    if (idx + start_index) % 50 == 0:
        save_progress(prev_weather_list, "prev_weather_partial.pkl")

extreme_df['prev_72h_weather'] = prev_weather_list
extreme_df.to_pickle("extreme_events_with_prev_weather.pkl")
logging.info("Final data saved to 'extreme_events_with_prev_weather.pkl'.")

if failed_rows:
    pd.DataFrame(failed_rows).to_csv("failed_rows.csv", index=False)
    logging.info("Some API calls failed. Details saved to 'failed_rows.csv'.")
else:
    logging.info("All API calls succeeded.")
```

```
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

```
import pickle
with open("prev_weather_partial.pkl", "rb") as f:
    prev_weather_list = pickle.load(f)
```

```
len(prev_weather_list)
```

```
import satlaspretrain_models
import torch
weights_manager = satlaspretrain_models.Weights()
```

```
model = weights_manager.get_pretrained_model(model_identifier="Sentinel2_SwinB_SI_RGB", fpn=Tr

# Expected input is a portion of a Sentinel-2 L1C TCI image.
# The 0-255 pixel values should be divided by 255 so they are 0-1.
# tensor = tci_image[None, :, :, :] / 255
tensor = torch.zeros((1, 3, 512, 512), dtype=torch.float32)

# Since we only loaded the backbone, it outputs feature maps from the Swin-v2-Base backbone.
output = model(tensor)
print([feature_map.shape for feature_map in output])
# [torch.Size([1, 128, 128, 128]), torch.Size([1, 256, 64, 64]), torch.Size([1, 512, 32, 32]),
```

combining all data

```
import pickle


extreme_events_10001_15000 = pickle.load(open("extreme_events_10001_15000.pkl", "rb"))
extreme_events_15001_19448 = pickle.load(open("extreme_events_15001_19448.pkl", "rb"))
extreme_events_19449_20000 = pickle.load(open("extreme_events_19449_20000.pkl", "rb"))
extreme_events_20001_30000 = pickle.load(open("extreme_events_with_prev_weather (1)_20001_3000
extreme_events_30001_40000 = pickle.load(open("extreme_events_with_prev_weather (30000 - 40000
```

```
%pip install --upgrade pandas
```

```
extreme_events_10001_15000.head()
```

```
import pandas as pd
import pickle

with open("extreme_events_with_prev_weather (1).pkl", "rb") as f:
    bhavya = pd.read_pickle(f)
```

```
bhavya
```

```
len(bhavya)
```

```
import pandas as pd
extreme_df = pd.read_csv("Final_prediction_dataset.csv")
```

```
print(len(extreme_events_10001_15000))
print(len(extreme_events_15001_19448))
print(len(extreme_events_19449_20000))
print(len(extreme_events_20001_30000))
print(len(extreme_events_30001_40000))
```

```
extreme_events_30001_40000
```

```
final_ds = pd.concat([extreme_events_10001_15000, extreme_events_15001_19448, extreme_events_1
```

```
final_ds.info()
```

```
final_ds.dropna(inplace=True)
```

```
final_ds.info()
```

```
final_ds.to_csv("final_ds.csv", index=False)
```

```
final_ds = pd.read_csv("final_ds.csv")
```

```
final_ds.head()
```

```

```

```
final_ds.head()
```

```
final_ds.iloc[0, :]
```

```
import pandas as pd
from datetime import datetime, timedelta
import os
from tqdm import tqdm
import io
from PIL import Image
import numpy as np
import requests

# NASA GIBS WMS endpoint for EPSG:4326 (geographic coordinates)
wms_url = "https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi?"

# Define relevant layers for extreme weather prediction
# layers = [
#     "IMERG_Precipitation_Rate",
#     "VIIRS_SNPP_Ice_Surface_Temp_Day",
#     "VIIRS_SNPP_Ice_Surface_Temp_Night",
#     "GHRSST_L4_MUR_Sea_Surface_Temperature",
#     "VIIRS_SNPP_Cirrus_Reflectance_SWIR_M11",
#     "VIIRS_SNPP_Cirrus_Reflectance_VIS_NIR",
#     "CALIPSO_Imaging_Infrared_Radiometer_Brightness_Temperature_Day_CH3"
# ]

layers = ["MODIS_Terra_CorrectedReflectance_TrueColor"]
```

```python
img_size = (1024, 1024)
output_dir = "satellite_images"
os.makedirs(output_dir, exist_ok=True)
metadata_records = []
margin = 10

def is_image_empty(img_data):
    img = Image.open(io.BytesIO(img_data))
    img_array = np.array(img)
    return np.all(img_array == 0)

def get_wms_image(layer, bbox, date):
    params = {
        'SERVICE': 'WMS',
        'VERSION': '1.3.0',
        'REQUEST': 'GetMap',
        'LAYERS': layer,
        'STYLES': '',
        'CRS': 'EPSG:4326',
        'BBOX': ','.join(map(str, bbox)),
        'WIDTH': img_size[0],
        'HEIGHT': img_size[1],
        'FORMAT': 'image/png',
        'TIME': date
    }
    response = requests.get(wms_url, params=params)
    if response.status_code == 200 and response.headers['Content-Type'] == 'image/png':
        return response.content
    else:
        print("Failed to retrieve image: {response.status_code}")
        raise Exception(f"Failed to retrieve image: {response.status_code}")

# Assuming df is your DataFrame with event data
for idx, row in tqdm(df.iterrows(), total=len(df), desc="Processing events"):
    min_lon = min(row["begin_lon"], row["end_lon"]) - margin
    max_lon = max(row["begin_lon"], row["end_lon"]) + margin
    min_lat = min(row["begin_lat"], row["end_lat"]) - margin
    max_lat = max(row["begin_lat"], row["end_lat"]) + margin
    bbox = [min_lat, min_lon, max_lat, max_lon]

    event_date = datetime.strptime(row['begin_date_utc'], "%Y-%m-%d").date()

    for days_before in range(3):  # 0, 1, 2 days before the event
        current_date = event_date - timedelta(days=days_before)
        date_str = current_date.strftime("%Y-%m-%d")

        for layer in layers:
            safe_event_type = row["event_type"].replace(" ", "_")
            safe_layer_name = layer.replace(" ", "_")
            filename = os.path.join(output_dir, f"event{idx}_{safe_event_type}_{safe_layer_nar

            try:
                img_data = get_wms_image(layer, bbox, date_str)
                if is_image_empty(img_data):
                    continue
```

```
                with open(filename, "wb") as f:
                    f.write(img_data)


                metadata_records.append({
                    "event_id": idx,
                    "event_type": row["event_type"],
                    "layer": layer,
                    "image_filename": os.path.basename(filename),
                    "download_time": datetime.utcnow().strftime("%Y-%m-%d"),
                    "image_date": date_str,
                    "days_before_event": days_before,
                    "bbox": bbox,
                    "event_date": event_date.strftime("%Y-%m-%d")
                })
            except Exception as e:
                print(f"Failed to download {layer} for event {idx} on {date_str}: {e}")

metadata_df = pd.DataFrame(metadata_records)
metadata_csv = os.path.join(output_dir, "satellite_images_metadata.csv")
metadata_df.to_csv(metadata_csv, index=False)
print(f"Saved metadata to {metadata_csv}")
```

```
from owslib.wms import WebMapService
from datetime import datetime, timedelta

def get_layer_info(layer_name, wms_url):
    wms = WebMapService(wms_url, version='1.3.0')
    layer = wms.contents[layer_name]

    time_info = {}
    if 'time' in layer.dimensions:
        time_dim = layer.dimensions['time']
        if time_dim['values']:
            time_range = time_dim['values'][0].split('/')
            if len(time_range) == 3:
                start_date = datetime.strptime(time_range[0], '%Y-%m-%d')
                end_date = datetime.strptime(time_range[1], '%Y-%m-%d')
                interval = time_range[2]

                time_info['start_date'] = start_date.strftime('%Y-%m-%d')
                time_info['end_date'] = end_date.strftime('%Y-%m-%d')
                time_info['interval'] = interval

                if interval == 'P1D':
                    time_info['resolution'] = 'Daily'
                elif interval == 'P1M':
                    time_info['resolution'] = 'Monthly'
                elif interval == 'PT1H':
                    time_info['resolution'] = 'Hourly'
                else:
                    time_info['resolution'] = f'Custom ({interval})'

    return time_info
```

```python
# Example usage
wms_url = "https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi"
layers = [
    "MERRA2_2m_Air_Temperature_Monthly",
    "IMERG_Precipitation_Rate",
    "MERRA2_Relative_Humidity_After_Moist_700hPa_Monthly",
    "MERRA2_Surface_Wind_Speed_Monthly",
    "MODIS_Aqua_Cloud_Fraction_Day",
    "MODIS_Terra_Aerosol",
    "MODIS_Aqua_L3_SST_Thermal_9km_Day_Monthly",
    "SMAP_L3_Passive_Enhanced_Day_Soil_Moisture",
    "MODIS_Terra_L3_NDVI_Monthly"
]

for layer_name in layers:
    info = get_layer_info(layer_name, wms_url)
    print(f"Layer: {layer_name}")
    if info:
        print(f"  Resolution: {info['resolution']}")
        print(f"  Date Range: {info['start_date']} to {info['end_date']}")
    else:
        print("  No time information available")
    print()
```

```python
from owslib.wms import WebMapService
from datetime import datetime

# Define the WMS URL
wms_url = "https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi"
wms = WebMapService(wms_url, version="1.3.0")

# Define the years to check for data
years_to_check = [2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]

# Function to check if daily or hourly data is available for all specified years
def check_data_availability(layer_name):
    layer = wms.contents[layer_name]
    if 'time' in layer.dimensions:
        time_dim = layer.dimensions['time']
        if time_dim['values']:
            time_range = time_dim['values'][0].split('/')
            if len(time_range) == 3:
                start_date = datetime.strptime(time_range[0][:10], '%Y-%m-%d')
                end_date = datetime.strptime(time_range[1][:10], '%Y-%m-%d')
                interval = time_range[2]
                if interval in ['P1D', 'PT1H']:  # Check if interval is daily or hourly
                    if all(start_date.year <= year <= end_date.year for year in years_to_check
                        return interval
    return None

# Check each layer and print those with daily or hourly data for all specified years
print("Layers with daily or hourly data available for all specified years:")
for layer_name in wms.contents:
    availability = check_data_availability(layer_name)
```

```
    if availability:
        print(f"{layer_name}: {'Daily' if availability == 'P1D' else 'Hourly'}")
```

```
final_ds["event_type"].unique()
```

```
import pandas as pd
df = pd.read_csv("final_ds.csv")
df.info()
```

```
df["extreme"].value_counts()
```

```
df.info()
```

```
import pandas as pd
df = pd.read_csv("final_ds.csv")
df = df.iloc[:7000, :]
```

```
df.info()
```

```
import os
import pickle
import pandas as pd
import numpy as np
from PIL import Image
import io
import requests
from datetime import datetime, timedelta
from concurrent.futures import ThreadPoolExecutor, as_completed
import multiprocessing
from tqdm import tqdm

# Load the dataset
df = pd.read_csv("final_ds.csv")
df = df.iloc[:7000, :]
# NASA GIBS WMS endpoint for EPSG:4326
wms_url = "https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi?"

# Define relevant layers for extreme weather prediction
layers = [
    "IMERG_Precipitation_Rate",
    "VIIRS_SNPP_Ice_Surface_Temp_Day",
    "VIIRS_SNPP_Ice_Surface_Temp_Night",
    "GHRSST_L4_MUR_Sea_Surface_Temperature",
    "VIIRS_SNPP_Cirrus_Reflectance_SWIR_M11",
    "VIIRS_SNPP_Cirrus_Reflectance_VIS_NIR",
    "CALIPSO_Imaging_Infrared_Radiometer_Brightness_Temperature_Day_CH3",
    "MODIS_Terra_CorrectedReflectance_TrueColor"
]

# Image size and output directory
img_size = (512, 512)
output_dir = "satellite_images"
os.makedirs(output_dir, exist_ok=True)
margin = 3
```

```python
# Checkpoint file to track processed events
checkpoint_file = "processed_events.pkl"
if os.path.exists(checkpoint_file):
    with open(checkpoint_file, "rb") as f:
        processed_events = pickle.load(f)
else:
    processed_events = set()

def is_image_empty(img_data):
    """Check if the downloaded image is empty (all pixels are the same)."""
    img = Image.open(io.BytesIO(img_data))
    img_array = np.array(img)
    return np.all(img_array == 0) or np.all(img_array == 255)

def get_wms_image(layer, bbox, date):
    """Retrieve an image from the WMS service for a specific layer, bounding box, and date."""
    params = {
        'SERVICE': 'WMS',
        'VERSION': '1.3.0',
        'REQUEST': 'GetMap',
        'LAYERS': layer,
        'STYLES': '',
        'CRS': 'EPSG:4326',
        'BBOX': ','.join(map(str, bbox)),
        'WIDTH': img_size[0],
        'HEIGHT': img_size[1],
        'FORMAT': 'image/png',
        'TIME': date,
        'TRANSPARENT': False
    }
    response = requests.get(wms_url, params=params)
    if response.status_code == 200 and response.headers['Content-Type'] == 'image/png':
        return response.content
    else:
        return None

def process_event(idx, row, processed_events):
    """
    Process a single event row:
      - Calculate the bounding box with margin.
      - For each of 3 days (0, 1, 2 days before the event), and for each layer,
        download the image, check if it's empty, save it, and record metadata.
    """
    if idx in processed_events:
        return []  # Skip already processed events

    local_metadata = []
    # Compute the bounding box based on event coordinates
    min_lon = min(row["begin_lon"], row["end_lon"]) - margin
    max_lon = max(row["begin_lon"], row["end_lon"]) + margin
    min_lat = min(row["begin_lat"], row["end_lat"]) - margin
    max_lat = max(row["begin_lat"], row["end_lat"]) + margin
    bbox = [min_lat, min_lon, max_lat, max_lon]

    event_date = datetime.strptime(row['begin_date_utc'], "%Y-%m-%d").date()
```

```
    # Loop through the three days before the event
    for days_before in range(3):
        current_date = event_date - timedelta(days=days_before)
        date_str = current_date.strftime("%Y-%m-%d")

        for layer in layers:
            safe_event_type = row["event_type"].replace(" ", "_")
            safe_layer_name = layer.replace(" ", "_")
            filename = os.path.join(output_dir, f"event{idx}_{safe_event_type}_{safe_layer_nar
            try:
                img_data = get_wms_image(layer, bbox, date_str)
                if img_data is None or is_image_empty(img_data):
                    continue
                with open(filename, "wb") as f:
                    f.write(img_data)
                local_metadata.append({
                    "event_id": idx,
                    "event_type": row["event_type"],
                    "layer": layer,
                    "image_filename": os.path.basename(filename),
                    "download_time": datetime.utcnow().strftime("%Y-%m-%d"),
                    "image_date": date_str,
                    "days_before_event": days_before,
                    "bbox": bbox,
                    "event_date": event_date.strftime("%Y-%m-%d")
                })
            except Exception as e:
                print(f"Error processing event {idx}, layer {layer}, date {date_str}: {e}")
                continue

    # After successfully processing the event
    processed_events.add(idx)
    with open(checkpoint_file, "wb") as f:
        pickle.dump(processed_events, f)

    return local_metadata

# Main execution block
all_metadata = []
num_cores = multiprocessing.cpu_count()

# Use ThreadPoolExecutor for parallel processing of events
with ThreadPoolExecutor(max_workers=num_cores) as executor:
    futures = {
        executor.submit(process_event, idx, row, processed_events): idx
        for idx, row in df.iterrows() if idx not in processed_events
    }
    for future in tqdm(as_completed(futures), total=len(futures), desc="Processing events"):
        try:
            result = future.result()
            all_metadata.extend(result)
        except Exception as exc:
            print(f"Event processing generated an exception: {exc}")

# Combine new metadata with existing metadata, if any
```

```python
metadata_csv = os.path.join(output_dir, "satellite_images_metadata.csv")
if os.path.exists(metadata_csv):
    existing_metadata_df = pd.read_csv(metadata_csv)
    metadata_df = pd.DataFrame(all_metadata)
    combined_metadata_df = pd.concat([existing_metadata_df, metadata_df], ignore_index=True)
else:
    combined_metadata_df = pd.DataFrame(all_metadata)

# Save combined metadata to CSV
combined_metadata_df.to_csv(metadata_csv, index=False)
print(f"Saved metadata to {metadata_csv}")
```

```python
import os
import pickle
import pandas as pd
import numpy as np
from PIL import Image
import io
import requests
from datetime import datetime, timedelta
from concurrent.futures import ThreadPoolExecutor, as_completed
import multiprocessing
from tqdm import tqdm

# Load the dataset
df = pd.read_csv("final_ds.csv")
df = df.iloc[21000:27999, :]
# NASA GIBS WMS endpoint for EPSG:4326
wms_url = "https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi?"

# Define relevant layers for extreme weather prediction
layers = [
    "IMERG_Precipitation_Rate",
    "VIIRS_SNPP_Ice_Surface_Temp_Day",
    "VIIRS_SNPP_Ice_Surface_Temp_Night",
    "GHRSST_L4_MUR_Sea_Surface_Temperature",
    "VIIRS_SNPP_Cirrus_Reflectance_SWIR_M11",
    "VIIRS_SNPP_Cirrus_Reflectance_VIS_NIR",
    "CALIPSO_Imaging_Infrared_Radiometer_Brightness_Temperature_Day_CH3",
    "MODIS_Terra_CorrectedReflectance_TrueColor"
]

# Image size and output directory
img_size = (512, 512)
output_dir = "satellite_images"
os.makedirs(output_dir, exist_ok=True)
margin = 3

# Checkpoint file to track processed events
checkpoint_file = "processed_events.pkl"
if os.path.exists(checkpoint_file):
    with open(checkpoint_file, "rb") as f:
        processed_events = pickle.load(f)
else:
    processed_events = set()
```

```python
def is_image_empty(img_data):
    """Check if the downloaded image is empty (all pixels are the same)."""
    img = Image.open(io.BytesIO(img_data))
    img_array = np.array(img)
    return np.all(img_array == 0) or np.all(img_array == 255)

def get_wms_image(layer, bbox, date):
    """Retrieve an image from the WMS service for a specific layer, bounding box, and date."""
    params = {
        'SERVICE': 'WMS',
        'VERSION': '1.3.0',
        'REQUEST': 'GetMap',
        'LAYERS': layer,
        'STYLES': '',
        'CRS': 'EPSG:4326',
        'BBOX': ','.join(map(str, bbox)),
        'WIDTH': img_size[0],
        'HEIGHT': img_size[1],
        'FORMAT': 'image/png',
        'TIME': date,
        'TRANSPARENT': False
    }
    response = requests.get(wms_url, params=params)
    if response.status_code == 200 and response.headers['Content-Type'] == 'image/png':
        return response.content
    else:
        return None

def process_event(idx, row, processed_events):
    """
    Process a single event row:
      - Calculate the bounding box with margin.
      - For each of 3 days (0, 1, 2 days before the event), and for each layer,
        download the image, check if it's empty, save it, and record metadata.
    """
    if idx in processed_events:
        return []  # Skip already processed events

    local_metadata = []
    # Compute the bounding box based on event coordinates
    min_lon = min(row["begin_lon"], row["end_lon"]) - margin
    max_lon = max(row["begin_lon"], row["end_lon"]) + margin
    min_lat = min(row["begin_lat"], row["end_lat"]) - margin
    max_lat = max(row["begin_lat"], row["end_lat"]) + margin
    bbox = [min_lat, min_lon, max_lat, max_lon]

    event_date = datetime.strptime(row['begin_date_utc'], "%Y-%m-%d").date()

    # Loop through the three days before the event
    for days_before in range(3):
        current_date = event_date - timedelta(days=days_before)
        date_str = current_date.strftime("%Y-%m-%d")

        for layer in layers:
            safe_event_type = row["event_type"].replace(" ", "_")
            safe_layer_name = layer.replace(" ", "_")
```

```python
            filename = os.path.join(output_dir, f"event{idx}_{safe_event_type}_{safe_layer_nar
            try:
                img_data = get_wms_image(layer, bbox, date_str)
                if img_data is None or is_image_empty(img_data):
                    continue
                with open(filename, "wb") as f:
                    f.write(img_data)
                local_metadata.append({
                    "event_id": idx,
                    "event_type": row["event_type"],
                    "layer": layer,
                    "image_filename": os.path.basename(filename),
                    "download_time": datetime.utcnow().strftime("%Y-%m-%d"),
                    "image_date": date_str,
                    "days_before_event": days_before,
                    "bbox": bbox,
                    "event_date": event_date.strftime("%Y-%m-%d")
                })
            except Exception as e:
                print(f"Error processing event {idx}, layer {layer}, date {date_str}: {e}")
                continue

    # After successfully processing the event
    processed_events.add(idx)
    with open(checkpoint_file, "wb") as f:
        pickle.dump(processed_events, f)

    return local_metadata

# Main execution block
all_metadata = []
num_cores = multiprocessing.cpu_count()

# Use ThreadPoolExecutor for parallel processing of events
with ThreadPoolExecutor(max_workers=num_cores) as executor:
    futures = {
        executor.submit(process_event, idx, row, processed_events): idx
        for idx, row in df.iterrows() if idx not in processed_events
    }
    for future in tqdm(as_completed(futures), total=len(futures), desc="Processing events"):
        try:
            result = future.result()
            all_metadata.extend(result)
        except Exception as exc:
            print(f"Event processing generated an exception: {exc}")

# Combine new metadata with existing metadata, if any
metadata_csv = os.path.join(output_dir, "satellite_images_metadata.csv")
if os.path.exists(metadata_csv):
    existing_metadata_df = pd.read_csv(metadata_csv)
    metadata_df = pd.DataFrame(all_metadata)
    combined_metadata_df = pd.concat([existing_metadata_df, metadata_df], ignore_index=True)
else:
    combined_metadata_df = pd.DataFrame(all_metadata)

# Save combined metadata to CSV
```

```
combined_metadata_df.to_csv(metadata_csv, index=False)
print(f"Saved metadata to {metadata_csv}")
```

```
import os

folder_path = "satellite_images"  # Replace with your folder path
file_count = len([f for f in os.listdir(folder_path) if os.path.isfile(os.path.join(folder_pat
print(f"Number of files in {folder_path}: {file_count}")
```

Integrating final dataset

```
import pandas as pd
metdata = pd.read_csv("satellite_images/satellite_images_metadata.csv")
metdata.info()
```

```
metdata.head()
```

```
from PIL import Image
import numpy as np
import io
from tqdm import tqdm
def is_image_empty(img_data):
    """Check if the downloaded image is empty (all pixels are the same)."""
    img = Image.open(io.BytesIO(img_data))
    img_array = np.array(img)
    return np.all(img_array == 0) or np.all(img_array == 255)
```

```
import os
import pandas as pd
from PIL import Image
import numpy as np
import io
from tqdm import tqdm

# Define relevant layers for extreme weather prediction
layers = [
    "IMERG_Precipitation_Rate",
    "VIIRS_SNPP_Ice_Surface_Temp_Day",
    "VIIRS_SNPP_Ice_Surface_Temp_Night",
    "GHRSST_L4_MUR_Sea_Surface_Temperature",
    "VIIRS_SNPP_Cirrus_Reflectance_SWIR_M11",
    "VIIRS_SNPP_Cirrus_Reflectance_VIS_NIR",
    "CALIPSO_Imaging_Infrared_Radiometer_Brightness_Temperature_Day_CH3",
    "MODIS_Terra_CorrectedReflectance_TrueColor"
]

# Function to check if an image is empty (fully black or white)
def is_image_empty(img_path):
    """Check if an image is empty (all pixels are either 0 or 255)."""
    try:
        with Image.open(img_path) as img:
            img_array = np.array(img)
            return np.all(img_array == 0) or np.all(img_array == 255)
```

```python
    except Exception as e:
        print(f"Error processing {img_path}: {e}")
        return True  # Treat unreadable images as empty

# Load the final dataset
final_dataset = pd.read_csv("final_ds.csv")  # Replace with your actual dataset filename

# Folder where images are stored
image_folder = "satellite_images"  # Replace with your actual image folder

# Count empty image rows
empty_rows_count = 0
valid_image_counts = []  # Store number of valid images for each row

for _, row in tqdm(final_dataset.iterrows(), total=len(final_dataset)):
    empty_count = 0
    valid_count = 0

    for layer in layers:
        image_filename = f"event{row.name}_{row['event_type']}_{layer}.png"  # Modify based or
        image_path = os.path.join(image_folder, image_filename)

        if os.path.isfile(image_path):
            if is_image_empty(image_path):
                empty_count += 1
            else:
                valid_count += 1
        else:
            empty_count += 1  # If file is missing, treat it as empty

    if valid_count == 0:  # If all images are empty for this row
        empty_rows_count += 1
    else:
        valid_image_counts.append(valid_count)

# Print results
print(f"Total rows where all images are empty: {empty_rows_count}")
print(f"Distribution of valid images in remaining rows:")
print(pd.Series(valid_image_counts).value_counts().sort_index())
```

```python
import os
import pandas as pd
from PIL import Image
import numpy as np
import io
from tqdm import tqdm

# Function to check if an image is empty (all pixels are either 0 or 255)
def is_image_empty(img_path):
    """Check if the image is empty (all pixels are the same)."""
    try:
        with Image.open(img_path) as img:
            img_array = np.array(img)
            return np.all(img_array == 0) or np.all(img_array == 255)
    except Exception as e:
        print(f"Error processing {img_path}: {e}")
```

```
        return True  # Consider unreadable images as empty

# Load the metadata DataFrame
df = pd.read_csv("satellite_images/satellite_images_metadata.csv")  # Replace with the actual

# Folder where images are stored
image_folder = "satellite_images"  # Replace with your actual image folder

# Iterate through the DataFrame and count empty images
empty_count = 0
total_count = 0

for filename in tqdm(df["image_filename"]):
    image_path = os.path.join(image_folder, filename)

    if os.path.isfile(image_path):
        total_count += 1
        if is_image_empty(image_path):
            empty_count += 1
    else:
        print(f"File not found: {image_path}")

print(f"Total valid images checked: {total_count}")
print(f"Number of empty images: {empty_count}")
```

```
df.iloc[0,:]["image_filename"]
```

```
final_dataset.head()
```

```
import os
from tqdm import tqdm

# Function to check if an image is empty (all pixels are either 0 or 255)
def is_image_empty(img_path, threshold=0.70):
    """
    Check if an image is empty or non-informative.
    Args:
        img_path: Path to the image file
        threshold: Percentage of same-colored pixels to consider image empty (0.0 to 1.0)
    Returns:
        bool: True if image is considered empty/non-informative
    """
    try:
        with Image.open(img_path) as img:
            # Convert to grayscale to simplify analysis
            gray_img = img.convert('L')
            img_array = np.array(gray_img)

            # Check if image is predominantly white
            white_ratio = np.sum(img_array > 250) / img_array.size
            # Check if image is predominantly black
            black_ratio = np.sum(img_array < 5) / img_array.size

            return white_ratio > threshold or black_ratio > threshold
```

```python
    except Exception as e:
        print(f"Error processing {img_path}: {e}")
        return True  # Consider unreadable images as empty

def is_image_informative(img_path, threshold=0.05):
    """
    Check if the image contains sufficient non-white (informative) content.

    Parameters:
    - img_path: str, path to the image file.
    - threshold: float, proportion of non-white pixels required to consider the image informat

    Returns:
    - bool: True if the image is informative, False otherwise.
    """
    try:
        with Image.open(img_path) as img:
            # Convert image to grayscale
            gray_img = img.convert('L')
            img_array = np.array(gray_img)

            # Calculate the number of non-blank pixels (pixels that are not white)
            non_blank_pixels = np.sum(img_array < 255)  # Changed from '== 0' to '< 255'
            total_pixels = img_array.size

            # Calculate the proportion of non-blank pixels
            proportion_non_blank = non_blank_pixels / total_pixels

            # Determine if the image is informative based on the threshold
            return proportion_non_blank >= threshold

    except Exception as e:
        print(f"Error processing {img_path}: {e}")
        return False  # Consider unreadable images as non-informative

# Load the final dataset
# Folder where images are stored
image_folder = "satellite_images"

# Iterate through all files in the folder and count empty images
informative = 0
total_image_count = 0

for filename in tqdm(os.listdir(image_folder)):
    image_path = os.path.join(image_folder, filename)
    if os.path.isfile(image_path):
        total_image_count += 1
        if is_image_empty(image_path):
            informative += 1

print(f"Total images checked: {total_image_count}")
print(f"Number of empty images: {informative}")
```

```python
metdata.head()
```

```python
import pandas as pd

# Load the final dataset
final_df = pd.read_csv("final_ds.csv")

# Load the metadata CSV containing image filenames
metadata_df = pd.read_csv("satellite_images/satellite_images_metadata.csv")

# If your final_df does not have an explicit event identifier,
# add one based on its index.
if 'event_id' not in final_df.columns:
    final_df["event_id"] = final_df.index

# Group metadata by event_id and aggregate image filenames into a list.
filenames_by_event = metadata_df.groupby("event_id")["image_filename"].apply(list).reset_index
filenames_by_event.rename(columns={"image_filename": "filenames"}, inplace=True)
# print(filenames_by_event)
# Merge the aggregated filenames with the final dataset.
merged_df = pd.merge(final_df, filenames_by_event, on="event_id", how="left")
merged_df.head()
# # Optionally, if you don't need the event_id column, you can drop it:
# # merged_df.drop("event_id", axis=1, inplace=True)

# # Save the merged DataFrame to a new CSV
# merged_df.to_csv("final_ds_with_filenames.csv", index=False)

# print("Merged dataset saved as final_ds_with_filenames.csv")
```

```python
%pip install pandas
```

```python
metadata_df.shape
```

```python
final_ds.shape
```

```python
merged_df.shape
```

```python
merged_df[["begin_date_utc","prev_72h_weather"]]
```

```python
merged_df.iloc[0,:]["prev_72h_weather"]
```

```python
merged_df.iloc[0,:]["filenames"]
```

```python
# Function to check if an image is empty (all pixels are either 0 or 255)
def is_image_empty(img_path, threshold=0.70):
    """
    Check if an image is empty or non-informative.
    Args:
        img_path: Path to the image file
        threshold: Percentage of same-colored pixels to consider image empty (0.0 to 1.0)
    Returns:
        bool: True if image is considered empty/non-informative
    """
    try:
```

```
        with Image.open(img_path) as img:
            # Convert to grayscale to simplify analysis
            gray_img = img.convert('L')
            img_array = np.array(gray_img)

            # Check if image is predominantly white
            white_ratio = np.sum(img_array > 250) / img_array.size
            # Check if image is predominantly black
            black_ratio = np.sum(img_array < 5) / img_array.size

            return white_ratio > threshold or black_ratio > threshold

    except Exception as e:
        print(f"Error processing {img_path}: {e}")
        return True  # Consider unreadable images as empty

def count_empty_images_by_layer(image_folder, layers):
    """
    Count empty images for each layer in the dataset.

    Args:
        image_folder: Path to the folder containing images
        layers: List of layer names to analyze
    """
    # Dictionary to store counts of empty images for each layer
    empty_counts = {layer: 0 for layer in layers}
    total_counts = {layer: 0 for layer in layers}

    # Iterate through all images in the folder
    for filename in os.listdir(image_folder):
        if filename.endswith('.png'):
            # Find which layer this image belongs to
            for layer in layers:
                if layer in filename:
                    img_path = os.path.join(image_folder, filename)
                    total_counts[layer] += 1

                    if is_image_empty(img_path):
                        empty_counts[layer] += 1
                    break

    # Print results sorted by number of empty images
    print("\nEmpty Image Statistics by Layer:")
    print("-" * 50)
    sorted_layers = sorted(layers, key=lambda x: empty_counts[x], reverse=True)

    for layer in sorted_layers:
        if total_counts[layer] > 0:
            empty_ratio = (empty_counts[layer] / total_counts[layer]) * 100
            print(f"{layer}:")
            print(f"  Empty images: {empty_counts[layer]}/{total_counts[layer]} ({empty_ratio

# Usage
image_folder = "satellite_images"  # Replace with your image folder path
count_empty_images_by_layer(image_folder, layers)
```

```
merged_df["extreme"].value_counts()
```

```
merged_df = merged_df.iloc[:7000,:]
```

```
merged_df
```

```
merged_df.columns
```

```
import re
import pandas as pd

def func(s):
    if not isinstance(s, str):
        return []  # Return empty list for non-string inputs

    dict_pattern = r'\{[^}]+\}'
    dict_matches = re.findall(dict_pattern, s)
    kv_pattern = r"'([^']+)':\s([^,}]+)"
    extracted_data = []

    for dict_str in dict_matches[:10]:  # Limit to first 10 dictionaries
        kv_matches = re.findall(kv_pattern, dict_str)
        extracted_dict = {key: value.strip() for key, value in kv_matches}
        extracted_data.append(extracted_dict)

    return extracted_data
result = []
# Read the CSV in chunks
chunk_size = 10000  # Adjust this based on your available memory
for chunk in pd.read_csv("final_ds_with_filenames.csv", chunksize=chunk_size):
    chunk["prev_72h_weather"] = chunk['prev_72h_weather'].apply(func)
    result.append(chunk)
    # Process or save the chunk here
    print("Processed chunk")
merged_df = pd.concat(result, ignore_index=True)
print("All chunks processed")
```

```

```

```
merged_df[["begin_date_utc","prev_72h_weather"]]
```

```
def expand_weather_data(row):
    weather_data = row['prev_72h_weather']
    for key in weather_data[0].keys():
        if key != 'date':
            row[f'prev_72h_{key}'] = [entry[key] for entry in weather_data]
    return row

merged_df = merged_df.apply(expand_weather_data, axis=1)
```

```
len(merged_df.columns)
```

```python
import ast

def clean_and_filter_layers(df, column_name='filename'):
    # Step 1: Convert string representation of list to actual list if needed
    def parse_if_string(x):
        if isinstance(x, str):
            try:
                return ast.literal_eval(x)
            except:
                return x
        return x

    # Apply parsing if the column contains string representations of lists
    df[column_name] = df[column_name].apply(parse_if_string)

    # Step 2: Filter for specific layers
    patterns = [
        'VIIRS_SNPP_Cirrus_Reflectance_SWIR_M11',
        'VIIRS_SNPP_Cirrus_Reflectance_VIS_NIR',
        'MODIS_Terra_CorrectedReflectance_TrueColor'
    ]

    # If the column contains lists, we need to filter elements within each list
    def filter_specific_layers(file_list):
        if isinstance(file_list, list):
            return [f for f in file_list if any(pattern in f for pattern in patterns)]
        return file_list

    df[column_name] = df[column_name].apply(filter_specific_layers)

    # Remove rows where the filtered list is empty
    df = df[df[column_name].apply(lambda x: len(x) > 0 if isinstance(x, list) else True)]

    return df

filtered_df = clean_and_filter_layers(merged_df, column_name='filenames')
```

```python
filtered_df = filtered_df.iloc[:7000,:]
```

```python
filtered_df.shape
```

```python
from PIL import Image
import numpy as np
from pathlib import Path

# Function to check if an image is empty (all pixels are either 0 or 255)
def is_image_empty(img_path, threshold=0.70):
    """
    Check if an image is empty or non-informative.
    Args:
        img_path: Path to the image file
        threshold: Percentage of same-colored pixels to consider image empty (0.0 to 1.0)
    Returns:
        bool: True if image is considered empty/non-informative
```

```python
    """
    try:
        with Image.open(img_path) as img:
            # Convert to grayscale to simplify analysis
            gray_img = img.convert('L')
            img_array = np.array(gray_img)

            # Check if image is predominantly white
            white_ratio = np.sum(img_array > 250) / img_array.size
            # Check if image is predominantly black
            black_ratio = np.sum(img_array < 5) / img_array.size

            return white_ratio > threshold or black_ratio > threshold

    except Exception as e:
        print(f"Error processing {img_path}: {e}")
        return True  # Consider unreadable images as empty

def check_and_filter_empty_images(df, base_path, column_name='filenames', threshold=0.70):
    """
    Check filtered images for emptiness and remove rows containing any empty images.

    Args:
        df: DataFrame containing lists of image filenames
        base_path: Base directory path where images are stored
        column_name: Name of the column containing image filenames
        threshold: Threshold for determining empty images

    Returns:
        DataFrame with rows removed where any image in the list is empty
    """
    def check_image_list(image_list):
        for img_name in image_list:
            img_path = Path(base_path) / img_name
            if is_image_empty(img_path, threshold):
                return False  # If any image is empty, return False
        return True  # All images are valid

    # Apply the check to each row and keep only rows where all images are valid
    mask = df[column_name].apply(check_image_list)
    filtered_df = df[mask]

    # Print statistics
    removed_count = len(df) - len(filtered_df)
    print(f"Removed {removed_count} rows containing empty images")
    print(f"Remaining rows: {len(filtered_df)}")

    return filtered_df

# Usage example:
```

```python
# Assuming you have already filtered for the three specific layers
base_path = "satellite_images"  # Replace with your actual path
final_df = check_and_filter_empty_images(filtered_df, base_path)
```

```
final_df["extreme"].value_counts()
```

```
final_df.head()
```

```
final_df.iloc[1,:]
```

```
final_df.iloc[2,:]
```

```
final_df.head()
```

```
final_df.info()
```

```
final_df.columns
```

```
import ast
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_s
from sklearn.utils.class_weight import compute_class_weight

# ------------------------------
# Step 1: Data Preprocessing
# ------------------------------
def preprocess_data(df):
    # Select relevant columns (excluding 'prev_72h_weather')
    feature_columns = [col for col in df.columns if col.startswith('prev_72h_') and col != 'pr

    def safe_eval(x):
        # If x is not a scalar and is iterable (but not a string), convert to list.
        if not np.isscalar(x) and not isinstance(x, str):
            try:
                return list(x)
            except Exception:
                return []
        # If x is scalar, check for NaN.
        if pd.isna(x):
            return []
        # If x is a string, try to literal_eval it.
        if isinstance(x, str):
            x = x.strip()
            if x == "":
                return []
            try:
                return ast.literal_eval(x)
            except Exception as e:
                print(f"Failed to evaluate '{x}': {e}")
```

```python
                return []
        # Otherwise, if it's a number, wrap it in a list.
        return [x]

    def extract_numeric(x):
        values = safe_eval(x)
        if not values:
            return [np.nan]
        numeric_vals = []
        for val in values:
            try:
                numeric_vals.append(float(val))
            except Exception:
                numeric_vals.append(np.nan)
        return numeric_vals

    # Apply extraction cellwise using applymap.
    processed = df[feature_columns].applymap(extract_numeric)

    # Expand each cell's list into separate columns.
    expanded_dfs = []
    for col in feature_columns:
        expanded = processed[col].apply(pd.Series)
        # Rename columns, e.g., "prev_72h_temperature_2m_0", "prev_72h_temperature_2m_1", etc.
        expanded = expanded.add_prefix(f"{col}_")
        expanded_dfs.append(expanded)

    # Concatenate all expanded columns horizontally.
    processed_df = pd.concat(expanded_dfs, axis=1)
    # Add target variable
    processed_df['extreme'] = df['extreme']

    print("Preprocessed DataFrame head:")
    print(processed_df.head())
    return processed_df

# ------------------------------
# Step 2: Compute Class Weights (if needed)
# ------------------------------
def compute_class_weights(y):
    class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(y), y=y)
    return dict(zip(np.unique(y), class_weights))

# ------------------------------
# Step 3: Define Models and Parameter Grids
# ------------------------------
models = {
    "Random Forest": {
        "estimator": RandomForestClassifier(random_state=42, class_weight='balanced'),
        "param_grid": {
            "n_estimators": [100, 200],
            "max_depth": [5, 10, None],
            "max_features": ["sqrt", "log2"]
        }
    },
    "Gradient Boosting": {
```

```python
        "estimator": GradientBoostingClassifier(random_state=42),
        "param_grid": {
            "n_estimators": [100, 200],
            "max_depth": [3, 5],
            "learning_rate": [0.05, 0.1]
        }
    },
    "Logistic Regression": {
        "estimator": LogisticRegression(max_iter=1000, random_state=42, class_weight='balanced
        "param_grid": {
            "C": [0.1, 1, 10],
            "penalty": ["l2"],
            "solver": ["lbfgs"]
        }
    },
    "SVC": {
        "estimator": SVC(probability=True, random_state=42, class_weight='balanced'),
        "param_grid": {
            "C": [0.1, 1, 10],
            "kernel": ["rbf", "linear"],
            "gamma": ["scale", "auto"]
        }
    },
    "KNN": {
        "estimator": KNeighborsClassifier(),
        "param_grid": {
            "n_neighbors": [3, 5, 7],
            "weights": ["uniform", "distance"]
        }
    }
}

# ------------------------------
# Step 4: Define a Function to Evaluate a Model Using TimeSeriesSplit
# ------------------------------
def evaluate_model(estimator, X, y, cv_splits=5):
    tscv = TimeSeriesSplit(n_splits=cv_splits)
    accs, precs, recs, f1s, aucs = [], [], [], [], []
    for train_idx, val_idx in tscv.split(X):
        X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
        y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]
        estimator.fit(X_train, y_train)
        y_pred = estimator.predict(X_val)
        try:
            y_pred_proba = estimator.predict_proba(X_val)[:, 1]
        except AttributeError:
            y_pred_proba = estimator.decision_function(X_val)
        accs.append(accuracy_score(y_val, y_pred))
        precs.append(precision_score(y_val, y_pred, pos_label=1, zero_division=0))
        recs.append(recall_score(y_val, y_pred, pos_label=1, zero_division=0))
        f1s.append(f1_score(y_val, y_pred, pos_label=1, zero_division=0))
        aucs.append(roc_auc_score(y_val, y_pred_proba))
    return {
        "Accuracy": np.mean(accs),
        "Precision": np.mean(precs),
        "Recall": np.mean(recs),
```

```
        "F1-score": np.mean(f1s),
        "AUC": np.mean(aucs)
    }

# ------------------------------
# Step 5: Main Script
# ------------------------------
if __name__ == '__main__':
    # Load your DataFrame (adjust this to your data source)
    # For example:
    # df = pd.read_csv("your_data.csv")
    # Here, we assume df is already defined.

    print("Original DataFrame shape:", df.shape)

    # Preprocess the data
    processed_df = preprocess_data(df)

    # Split features and target
    X = processed_df.drop('extreme', axis=1)
    y = processed_df['extreme']

    # Handle missing values:
    # Drop columns that are entirely NaN
    X = X.dropna(axis=1, how='all')
    # Impute remaining missing values with the mean
    from sklearn.impute import SimpleImputer
    imputer = SimpleImputer(strategy='mean')
    X_imputed = pd.DataFrame(imputer.fit_transform(X), columns=X.columns, index=X.index)

    # Normalize features
    scaler = StandardScaler()
    X_scaled = pd.DataFrame(scaler.fit_transform(X_imputed), columns=X_imputed.columns, index=

    # Optional: Compute and display class weights
    weights = compute_class_weights(y)
    print("Computed class weights:", weights)

    # To store final results for each model
    results = []

    # Loop over each model: run GridSearchCV with TimeSeriesSplit and evaluate best estimator
    for model_name, model_dict in models.items():
        print(f"\nTuning and evaluating model: {model_name}")
        estimator = model_dict["estimator"]
        param_grid = model_dict["param_grid"]
        tscv = TimeSeriesSplit(n_splits=5)
        grid = GridSearchCV(estimator, param_grid, cv=tscv, scoring='accuracy', n_jobs=-1, err
        try:
            grid.fit(X_scaled, y)
        except Exception as e:
            print(f"GridSearchCV failed for {model_name}: {e}")
            continue
        best_estimator = grid.best_estimator_
        print(f"Best parameters for {model_name}: {grid.best_params_}")
        metrics = evaluate_model(best_estimator, X_scaled, y, cv_splits=5)
```

```
        metrics["Model"] = model_name
        results.append(metrics)

    # Create a final results DataFrame and display it
    results_df = pd.DataFrame(results).set_index("Model")
    results_df = results_df[["Accuracy", "Precision", "Recall", "F1-score", "AUC"]]

    print("\nFinal Model Comparison Report:")
    print(results_df.round(4))
```

```
final_df[final_df["extreme"]==0]
```

```
final_df[final_df["extreme"]==1]
```

```
final_df.columns
```

```
final_df.info()
```

```
df.drop(columns=["prev_72h_weather"], inplace=True)
```

```
# Corrected Implementation for Multi-Class AUC Calculation
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Bidirectional, LSTM, Dense, Masking, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import TimeSeriesSplit
from sklearn.utils.class_weight import compute_class_weight
from imblearn.pipeline import Pipeline

# Custom Reshaper for Temporal Data
class TemporalReshaper:
    def fit(self, X, y=None): return self
    def transform(self, X): return X.reshape(X.shape[0], -1)
    def inverse_transform(self, X):
        return X.reshape(-1, 72, len(time_series_columns))

# Enhanced Data Preprocessing
def preprocess_data(df):
    time_series_columns = [col for col in df.columns if 'prev_72h_' in col]

    # Convert time series data to float32 arrays
    X = np.stack([df[col].apply(safe_eval).values for col in time_series_columns], axis=-1)

    # Handle NaNs with temporal-aware imputation
    for f_idx in range(X.shape[-1]):
        feature = X[..., f_idx]
        feature[np.isnan(feature)] = np.nanmean(feature)

    # Encode labels as one-hot vectors
    le = LabelEncoder()
    y = to_categorical(le.fit_transform(df['event_type']))
```

```
    return X.astype('float32'), y, le.classes_

# Build LSTM Model with AUC Compatibility
def build_model(input_shape, num_classes):
    model = Sequential([
        Masking(mask_value=np.nan, input_shape=input_shape),
        Bidirectional(LSTM(128, return_sequences=True, dropout=0.3)),
        Bidirectional(LSTM(64, dropout=0.2)),
        Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=Adam(learning_rate=1e-3),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# Custom AUC Callback for Multi-Class
from tensorflow.keras.callbacks import Callback
from sklearn.metrics import roc_auc_score

class MacroAUC(Callback):
    def __init__(self, X_val, y_val):
        super().__init__()
        self.X_val = X_val
        self.y_val = y_val.argmax(axis=1)  # Convert back to integer labels

    def on_epoch_end(self, epoch, logs=None):
        y_pred = self.model.predict(self.X_val)
        auc = roc_auc_score(self.y_val, y_pred, multi_class='ovo')
        print(f'\nValidation Macro AUC: {auc:.4f}')

# Main Execution Pipeline
def main(df):
    X, y, classes = preprocess_data(df)
    num_classes = len(classes)

    tscv = TimeSeriesSplit(n_splits=5)
    for fold, (train_idx, test_idx) in enumerate(tscv.split(X)):
        X_train, X_val = X[train_idx], X[test_idx]
        y_train, y_val = y[train_idx], y[test_idx]

        # Class balancing
        sample_weights = compute_class_weight('balanced', classes=np.unique(y_train.argmax(1))
        sample_weights = sample_weights[y_train.argmax(1)]

        model = build_model(X.shape[1:], num_classes)
        auc_callback = MacroAUC(X_val, y_val)

        model.fit(
            X_train, y_train,
            validation_data=(X_val, y_val),
            epochs=100,
            batch_size=64,
```

```
            sample_weight=sample_weights,
            callbacks=[auc_callback],
            verbose=1
        )

    return model

# Execute the corrected pipeline
final_model = main(df)
```

```
X.shape
```

```
%pip install imblearn
```

```

```

```
import pandas as pd

# Load the final dataset
final_df = pd.read_csv("final_ds.csv")

# Load the metadata CSV containing image filenames
metadata_df = pd.read_csv("satellite_images/satellite_images_metadata.csv")

# If your final_df does not have an explicit event identifier,
# add one based on its index.
if 'event_id' not in final_df.columns:
    final_df["event_id"] = final_df.index

# Group metadata by event_id and aggregate image filenames into a list.
filenames_by_event = metadata_df.groupby("event_id")["image_filename"].apply(list).reset_index
filenames_by_event.rename(columns={"image_filename": "filenames"}, inplace=True)
# print(filenames_by_event)
# Merge the aggregated filenames with the final dataset.
merged_df = pd.merge(final_df, filenames_by_event, on="event_id", how="left")
merged_df.head()
# # Optionally, if you don't need the event_id column, you can drop it:
# # merged_df.drop("event_id", axis=1, inplace=True)

# # Save the merged DataFrame to a new CSV
# merged_df.to_csv("final_ds_with_filenames.csv", index=False)

# print("Merged dataset saved as final_ds_with_filenames.csv")
```

```
merged_df = merged_df.iloc[:7000, :]
```

```
import re
import pandas as pd

def func(s):
    if not isinstance(s, str):
        return []  # Return empty list for non-string inputs

    dict_pattern = r'\{[^}]+\}'
```

```python
    dict_matches = re.findall(dict_pattern, s)
    kv_pattern = r"'([^']+)':\s([^,}]+)"
    extracted_data = []

    for dict_str in dict_matches:  # Limit to first 10 dictionaries
        kv_matches = re.findall(kv_pattern, dict_str)
        extracted_dict = {key: value.strip() for key, value in kv_matches}
        extracted_data.append(extracted_dict)

    return extracted_data
result = []
# Read the CSV in chunks
chunk_size = 10000  # Adjust this based on your available memory
for chunk in pd.read_csv("final_ds_with_filenames.csv", chunksize=chunk_size):
    chunk["prev_72h_weather"] = chunk['prev_72h_weather'].apply(func)
    result.append(chunk)
    # Process or save the chunk here
    print("Processed chunk")
merged_df = pd.concat(result, ignore_index=True)
merged_df = merged_df.iloc[:7000, :]
print("All chunks processed")
```

```python
len(merged_df["prev_72h_weather"].iloc[0])
```

```python
import tqdm

tqdm.tqdm.pandas()
def expand_weather_data(row):
    weather_data = row['prev_72h_weather']
    for key in weather_data[0].keys():
        if key != 'date':
            row[f'prev_72h_{key}'] = [entry[key] for entry in weather_data]
    return row

merged_df = merged_df.progress_apply(expand_weather_data, axis=1)
```

```python
import ast

def clean_and_filter_layers(df, column_name='filename'):
    # Step 1: Convert string representation of list to actual list if needed
    def parse_if_string(x):
        if isinstance(x, str):
            try:
                return ast.literal_eval(x)
            except:
                return x
        return x

    # Apply parsing if the column contains string representations of lists
    df[column_name] = df[column_name].progress_apply(parse_if_string)

    # Step 2: Filter for specific layers
    patterns = [
        'VIIRS_SNPP_Cirrus_Reflectance_SWIR_M11',
        'VIIRS_SNPP_Cirrus_Reflectance_VIS_NIR',
```

```python
        'MODIS_Terra_CorrectedReflectance_TrueColor'
    ]

    # If the column contains lists, we need to filter elements within each list
    def filter_specific_layers(file_list):
        if isinstance(file_list, list):
            return [f for f in file_list if any(pattern in f for pattern in patterns)]
        return file_list

    df[column_name] = df[column_name].progress_apply(filter_specific_layers)

    # Remove rows where the filtered list is empty
    df = df[df[column_name].progress_apply(lambda x: len(x) > 0 if isinstance(x, list) else Tr

    return df

filtered_df = clean_and_filter_layers(merged_df, column_name='filenames')
```

```python
filtered_df = filtered_df.iloc[:7000,:]
```

```python
filtered_df.info()
```

```python
from PIL import Image
import numpy as np
from pathlib import Path

# Function to check if an image is empty (all pixels are either 0 or 255)
def is_image_empty(img_path, threshold=0.70):
    """
    Check if an image is empty or non-informative.
    Args:
        img_path: Path to the image file
        threshold: Percentage of same-colored pixels to consider image empty (0.0 to 1.0)
    Returns:
        bool: True if image is considered empty/non-informative
    """
    try:
        with Image.open(img_path) as img:
            # Convert to grayscale to simplify analysis
            gray_img = img.convert('L')
            img_array = np.array(gray_img)

            # Check if image is predominantly white
            white_ratio = np.sum(img_array > 250) / img_array.size
            # Check if image is predominantly black
            black_ratio = np.sum(img_array < 5) / img_array.size

            return white_ratio > threshold or black_ratio > threshold

    except Exception as e:
        print(f"Error processing {img_path}: {e}")
        return True  # Consider unreadable images as empty

def check_and_filter_empty_images(df, base_path, column_name='filenames', threshold=0.70):
    """
```

```
    Check filtered images for emptiness and remove rows containing any empty images.

    Args:
        df: DataFrame containing lists of image filenames
        base_path: Base directory path where images are stored
        column_name: Name of the column containing image filenames
        threshold: Threshold for determining empty images

    Returns:
        DataFrame with rows removed where any image in the list is empty
    """
    def check_image_list(image_list):
        for img_name in image_list:
            img_path = Path(base_path) / img_name
            if is_image_empty(img_path, threshold):
                return False  # If any image is empty, return False
        return True  # All images are valid

    # Apply the check to each row and keep only rows where all images are valid
    mask = df[column_name].progress_apply(check_image_list)
    filtered_df = df[mask]

    # Print statistics
    removed_count = len(df) - len(filtered_df)
    print(f"Removed {removed_count} rows containing empty images")
    print(f"Remaining rows: {len(filtered_df)}")

    return filtered_df

# Usage example:
```

```
# Assuming you have already filtered for the three specific layers
base_path = "satellite_images"  # Replace with your actual path
final_df = check_and_filter_empty_images(filtered_df, base_path)
```

```
final_df["extreme"].value_counts()
```

```
final_df.to_csv("final_filtered_df.csv", index=False)
```

```
import pandas as pd
final_df = pd.read_csv("final_filtered_df.csv")
```

```
len(final_df["prev_72h_wind_direction_100m"].iloc[0].split(","))
```

```
final_df["filenames"].iloc[0]
```

EDA

```
import pandas as pd
import numpy as np
from scipy.stats import mannwhitneyu
import ast
```

```python
# Step 1: Clean the 'prev_' columns
def clean_prev_column(column):
    def safe_convert(value):
        try:
            if isinstance(value, str):
                return ast.literal_eval(value)
            elif isinstance(value, list):
                return value
            else:
                return []  # Replace invalid entries with an empty list
        except:
            return []  # Handle any parsing errors

    return column.apply(safe_convert)

# Clean all columns starting with 'prev_'
prev_columns = [col for col in df.columns if col.startswith('prev_')]
for col in prev_columns:
    df[col] = clean_prev_column(df[col])

# Step 2: Perform Mann-Whitney U test with balanced sampling
results = {}

for event in ['Flood', 'Heavy Rain', 'Flash Flood', 'Debris Flow']:
    results[event] = {}

    # Filter data for this event type
    event_df = df[df['event_type'] == event]
    extreme_event_df = event_df[event_df['extreme'] == 1]
    non_extreme_event_df = event_df[event_df['extreme'] == 0]

    print(f"\nEvent Type: {event}")

    for col in prev_columns:
        # Flatten lists into single arrays for comparison
        extreme_values = np.concatenate(extreme_event_df[col].values).astype(float)
        non_extreme_values = np.concatenate(non_extreme_event_df[col].values).astype(float)

        # Remove NaN or invalid values from both arrays
        extreme_values = extreme_values[~np.isnan(extreme_values)]
        non_extreme_values = non_extreme_values[~np.isnan(non_extreme_values)]

        # Balance sampling (sample equal sizes from both groups)
        min_size = min(len(extreme_values), len(non_extreme_values))
        if min_size > 0:
            extreme_sample = np.random.choice(extreme_values, min_size, replace=False)
            non_extreme_sample = np.random.choice(non_extreme_values, min_size, replace=False)

            # Perform Mann-Whitney U test (use exact method when appropriate)
            method = 'exact' if min_size < 50 else 'asymptotic'
            stat, p_value = mannwhitneyu(extreme_sample, non_extreme_sample, alternative='two-
            results[event][col] = {
                'p_value': p_value,
                'significant': p_value < 0.05
            }
```

```
        else:
            results[event][col] = {'p_value': None, 'significant': False}

# Step 3: Display significant results clearly
for event_type, cols in results.items():
    print(f"\nEvent Type: {event_type}")
    for col_name, result in cols.items():
        if result['significant']:
            print(f"Column: {col_name}, p-value: {result['p_value']:.4f} --> Significant diffe
        else:
            print(f"Column: {col_name}, p-value: {result['p_value']:.4f} --> No significant di
```

```
import pandas as pd
import numpy as np
import ast

from sklearn.model_selection import train_test_split, StratifiedKFold, cross_validate, GridSea
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (roc_auc_score, balanced_accuracy_score, f1_score,
                             precision_score, recall_score, confusion_matrix, make_scorer)
from sklearn.preprocessing import StandardScaler

# Import SMOTE from imblearn for oversampling
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as imbpipeline

# Assume 'df' is your DataFrame with 7000 rows.
df = pd.read_csv("final_filtered_df.csv")
df.drop(columns=["prev_72h_weather"], inplace=True)
extreme = df[df["extreme"] == 1]
non_extreme = df[df["extreme"] == 0].sample(437)
df = pd.concat([extreme, non_extreme])
columns_to_drop = [f'img_feat_{i}' for i in range(882)]
columns_to_drop.append('prev_72h_weather"')
# Drop the columns from the dataframe
df = df.drop(columns=columns_to_drop, errors='ignore')

# If you want to see the remaining columns

# If the 'prev_' columns are string representations of lists, convert them:
def safe_literal_eval(x):
    if isinstance(x, list):
        x = [float(i) for i in x]
        return x
    try:

        temp = ast.literal_eval(x)
        temp = [float(i) for i in temp]
        return temp

    except Exception as e:
        return np.nan

# Identify all columns starting with "prev_"
prev_cols = [col for col in df.columns if col.startswith('prev_')]
for col in prev_cols:
```

```
    df[col] = df[col].apply(safe_literal_eval)

# Aggregate each "prev_" column into summary statistics: mean, std, min, and max.
for col in prev_cols:
    df[f'{col}_mean'] = df[col].apply(lambda x: np.mean(x) if isinstance(x, list) else np.nan)
    df[f'{col}_std'] = df[col].apply(lambda x: np.std(x) if isinstance(x, list) else np.nan)
    df[f'{col}_min'] = df[col].apply(lambda x: np.min(x) if isinstance(x, list) else np.nan)
    df[f'{col}_max'] = df[col].apply(lambda x: np.max(x) if isinstance(x, list) else np.nan)

# Drop the raw list columns to simplify the data
df.drop(columns=prev_cols, inplace=True)

# Drop columns that are non-numeric or not needed for modeling
cols_to_drop = ['event_type', 'begin_date_time', 'cz_timezone', 'end_date_time',
                'begin_date_utc', 'begin_time_utc', 'end_date_utc', 'end_time_utc',
                'start_date_72h', 'end_date', 'event_datetime', 'prev_72h_weather',
                'filenames']
df_model = df.drop(columns=cols_to_drop, errors='ignore')

# Separate features (X) and target (y). Here, 'extreme' is the target.
X = df_model.drop(columns=['extreme', 'cluster', 'event_id'], errors='ignore')
y = df_model['extreme']

# Fill any missing values with the column mean.
X = X.fillna(X.mean())

# Split into training and test sets with stratification to maintain class distribution.
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    test_size=0.2, random_state=42)

# Build a pipeline that includes SMOTE for oversampling, scaling, and a classifier.
# We use imblearn's Pipeline to incorporate the sampler.
pipeline = imbpipeline([
    ('sampler', SMOTE(random_state=42)),
    ('scaler', StandardScaler()),
    ('clf', RandomForestClassifier(class_weight='balanced', random_state=42))
])

# Define scoring metrics
scoring = {
    'roc_auc': 'roc_auc',
    'balanced_accuracy': make_scorer(balanced_accuracy_score),
    'f1': 'f1',
    'precision': 'precision',
    'recall': 'recall'
}

# Use stratified 5-fold cross-validation
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_results = cross_validate(pipeline, X_train, y_train, cv=cv, scoring=scoring, return_train_s

print("Cross-validation Results with SMOTE:")
for key in sorted(cv_results.keys()):
    print(f"{key}: {np.mean(cv_results[key]):.4f}")

# Hyperparameter tuning via GridSearchCV (including resampling in the pipeline)
```

```
param_grid = {
    'clf__n_estimators': [100, 200],
    'clf__max_depth': [None, 10, 20]
}

grid = GridSearchCV(pipeline, param_grid, cv=cv, scoring='roc_auc', n_jobs=-1)
grid.fit(X_train, y_train)
print("\nBest hyperparameters with SMOTE:")
print(grid.best_params_)
print(f"Best CV ROC AUC: {grid.best_score_:.4f}")

# Evaluate the best model on the test set
best_model = grid.best_estimator_
y_pred = best_model.predict(X_test)
y_prob = best_model.predict_proba(X_test)[:, 1]

roc_auc = roc_auc_score(y_test, y_prob)
bal_acc = balanced_accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print("\nTest Set Evaluation Metrics with SMOTE:")
print(f"ROC AUC: {roc_auc:.4f}")
print(f"Balanced Accuracy: {bal_acc:.4f}")
print(f"F1 Score: {f1:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print("Confusion Matrix:")
print(cm)
```

```
import satlaspretrain_models
import torch

weights_manager = satlaspretrain_models.Weights()
model = weights_manager.get_pretrained_model(model_identifier="Sentinel2_SwinB_SI_RGB", device

# Load the model weights onto the CPU

model.eval()
```

```
import torch
from torchvision import transforms
from PIL import Image
import numpy as np
import pandas as pd
import ast
import logging
from tqdm.auto import tqdm
from concurrent.futures import ThreadPoolExecutor, as_completed
from tqdm.auto import tqdm

tqdm.pandas()  # Registers progress_apply for pandas
```

```python
# ==========================
# SET UP LOGGING
# ==========================
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')

# ==========================
# LOAD DATAFRAME
# ==========================
df = pd.read_csv("final_filtered_df.csv")
extreme = df[df["extreme"] == 1]
non_extreme = df[df["extreme"] == 0].sample(437)
df = pd.concat([extreme, non_extreme])

def safe_literal_eval(x):
    """Safely evaluate a string representation of a list."""
    if isinstance(x, list):
        return x
    try:
        return ast.literal_eval(x)
    except Exception:
        return []
df['filenames'] = df['filenames'].apply(safe_literal_eval)

# ==========================
# DEFINE IMAGE TRANSFORMATIONS
# ==========================
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])

# ==========================
# DEFINE THE ROW PROCESSING FUNCTION
# ==========================
def process_row(filenames):
    """
    Process a list of image filenames (for one row): load images,
    extract features from each, and concatenate them.
    """
    features = []
    for file in filenames:
        try:
            full_path = "satellite_images/" + file
            image = Image.open(full_path).convert('RGB')
            image = preprocess(image)
            image = image.unsqueeze(0)  # Add batch dimension
            with torch.no_grad():
                x = image
                row_features = []
                # Iterate over layers in your model's backbone
                for i, layer in enumerate(model.backbone.backbone.features):
                    x = layer(x)
```

```
                    if i in selected_layers:
                        pooled_feature = torch.mean(x, dim=(2, 3))
                        row_features.append(pooled_feature)
                if row_features:
                    concatenated_features = torch.cat(row_features, dim=1)
                    features.append(concatenated_features.squeeze(0).numpy())
        except Exception as e:
            logging.error(f"Error processing file {file}: {e}")
            features.append(np.zeros(98))
    if features:
        return np.concatenate(features)
    else:
        return np.zeros(98)

# ==========================
# LOAD YOUR MODEL
# ==========================
# Make sure your model is loaded and in evaluation mode.
# For example, using timm:
# import timm
# model = timm.create_model('swin_base_patch4_window7_224', pretrained=True)
# model.eval()

try:
    model
except NameError:
    raise NameError("Model is not defined. Please load your pre-trained model before running.'

selected_layers = [1, 3, 5]  # Adjust these indices as needed

# ==========================
# PARALLEL PROCESSING WITH THREADPOOL
# ==========================
logging.info("Starting parallel image feature extraction across rows using threads.")
results = [None] * len(df)  # Preallocate list for ordered results

with ThreadPoolExecutor(max_workers=8) as executor:  # Adjust max_workers if needed
    # Submit one task per row (each row's filenames list)
    future_to_index = {executor.submit(process_row, row): idx for idx, row in enumerate(df['fi
    for future in tqdm(as_completed(future_to_index), total=len(future_to_index), desc="Rows p
        idx = future_to_index[future]
        try:
            results[idx] = future.result()
        except Exception as e:
            logging.error(f"Error processing row {idx}: {e}")
            results[idx] = np.zeros(98)

df['image_features'] = results
logging.info("Completed parallel image feature extraction.")
```

```
df["image_features"].iloc[55].shape
```

```
df.to_csv("final_df_image_features.csv", index=False)
```

```python
import pandas as pd
temp = pd.read_csv("final_df_image_features.csv")
temp.drop(columns=["prev_72h_weather"], inplace=True)
temp.info()
```

```python
import numpy as np
import ast
temp.drop
def safe_literal_eval(x):
    if isinstance(x, list):
        x = [float(i) for i in x]
        return x
    try:
        temp = ast.literal_eval(x)
        temp = [float(i) for i in temp]
        return temp

    except Exception as e:
        print(e, x)
        return np.nan
prev_cols = [col for col in temp.columns if col.startswith('prev_')]
for col in prev_cols:
    temp[col] = temp[col].apply(safe_literal_eval)
```

```python
temp
```

```python
import numpy as np

# Assuming final_df_images_features["image_features"].iloc[0] contains the string
final_df
def apply_string_cleaning(array_str):
    # Clean the string by removing unwanted characters
    try:
        print(type(array_str))
        cleaned_str = array_str.replace('[', '').replace(']', '').replace('\n', ' ')
        array = np.fromstring(cleaned_str, sep=' ')
        return array
    except Exception as e:
        print(array_str)
# Clean the string by removing unwanted characters
final_df["images_features"] = final_df["images_features"].apply(apply_string_cleaning)
```

```python
tqdm.pandas()
# Expand the image features into separate columns (each feature as a column)
def expand_features(feat_array):
    if isinstance(feat_array, np.ndarray):
        return pd.Series(feat_array)
    else:
        return pd.Series()

df_image_features = df['image_features'].progress_apply(expand_features)
df_image_features.columns = [f'img_feat_{i}' for i in range(df_image_features.shape[1])]

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

```python
# 5. COMBINE IMAGE FEATURES WITH OTHER DATA
# ==========================
# Drop columns not needed for modeling
cols_to_drop = ['event_type', 'begin_date_time', 'cz_timezone', 'end_date_time',
                'begin_date_utc', 'begin_time_utc', 'end_date_utc', 'end_time_utc',
                'start_date_72h', 'end_date', 'event_datetime', 'prev_72h_weather',
                'filenames', 'image_features']
df_model = df.drop(columns=cols_to_drop, errors='ignore')

# Concatenate the expanded image features with the rest of the data
df_model = pd.concat([df_model, df_image_features], axis=1)
```

```python
def safe_literal_eval(x):
    if isinstance(x, list):
        x = [float(i) for i in x]
        return x
    try:
        temp = ast.literal_eval(x)
        temp = [float(i) for i in temp]
        return temp

    except Exception:
        return np.nan

# Identify all columns starting with "prev_"
prev_cols = [col for col in df_model.columns if col.startswith('prev_')]
for col in prev_cols:
    df_model[col] = df_model[col].apply(safe_literal_eval)
print(prev_cols)
print(df_model["prev_72h_temperature_2m"].values)
# Aggregate each "prev_" column into summary statistics: mean, std, min, and max.
for col in prev_cols:
    df_model[f'{col}_mean'] = df_model[col].apply(lambda x: np.mean(x) if isinstance(x, list)
    df_model[f'{col}_std'] = df_model[col].apply(lambda x: np.std(x) if isinstance(x, list) el
    df_model[f'{col}_min'] = df_model[col].apply(lambda x: np.min(x) if isinstance(x, list) el
    df_model[f'{col}_max'] = df_model[col].apply(lambda x: np.max(x) if isinstance(x, list) el

# Drop the raw list columns to simplify the data
df_model.drop(columns=prev_cols, inplace=True)

# Drop columns that are non-numeric or not needed for modeling
cols_to_drop = ['event_type', 'begin_date_time', 'cz_timezone', 'end_date_time',
                'begin_date_utc', 'begin_time_utc', 'end_date_utc', 'end_time_utc',
                'start_date_72h', 'end_date', 'event_datetime', 'prev_72h_weather',
                'filenames']
df_model = df_model.drop(columns=cols_to_drop, errors='ignore')

# Separate features (X) and target (y). Here, 'extreme' is the target.
X = df_model.drop(columns=['extreme', 'cluster', 'event_id'], errors='ignore')
y = df_model['extreme']

# Fill any missing values with the column mean.
X = X.fillna(X.mean())
```

```python
# ==========================
# 6. TRAINING WITH A SKLEARN/IMBLEARN PIPELINE
# ==========================
from sklearn.impute import SimpleImputer
from imblearn.pipeline import Pipeline as imbpipeline
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_validate, GridSea
from sklearn.metrics import (roc_auc_score, balanced_accuracy_score, f1_score,
                             precision_score, recall_score, confusion_matrix, make_scorer)

logging.info("Starting train-test split.")
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    test_size=0.2, random_state=42)
logging.info("Completed train-test split.")

# Build a pipeline for modeling
pipeline = imbpipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('sampler', SMOTE(random_state=42)),
    ('scaler', StandardScaler()),
    ('clf', RandomForestClassifier(class_weight='balanced', random_state=42))
])

scoring = {
    'roc_auc': 'roc_auc',
    'balanced_accuracy': make_scorer(balanced_accuracy_score),
    'f1': 'f1',
    'precision': 'precision',
    'recall': 'recall'
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
logging.info("Starting cross-validation.")
cv_results = cross_validate(pipeline, X_train, y_train, cv=cv, scoring=scoring, return_train_s
logging.info("Completed cross-validation.")

for key in sorted(cv_results.keys()):
    logging.info(f"{key}: {np.mean(cv_results[key]):.4f}")

param_grid = {
    'clf__n_estimators': [100, 200],
    'clf__max_depth': [None, 10, 20]
}

logging.info("Starting hyperparameter tuning with GridSearchCV.")
grid = GridSearchCV(pipeline, param_grid, cv=cv, scoring='roc_auc', n_jobs=-1)
grid.fit(X_train, y_train)
logging.info("Completed hyperparameter tuning.")

logging.info(f"Best hyperparameters: {grid.best_params_}")
logging.info(f"Best CV ROC AUC: {grid.best_score_:.4f}")

best_model = grid.best_estimator_
```

```
y_pred = best_model.predict(X_test)
y_prob = best_model.predict_proba(X_test)[:, 1]

roc_auc = roc_auc_score(y_test, y_prob)
bal_acc = balanced_accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

logging.info(f"Test Set ROC AUC: {roc_auc:.4f}")
logging.info(f"Test Set Balanced Accuracy: {bal_acc:.4f}")
logging.info(f"Test Set F1 Score: {f1:.4f}")
logging.info(f"Test Set Precision: {precision:.4f}")
logging.info(f"Test Set Recall: {recall:.4f}")
logging.info(f"Confusion Matrix:\n{cm}")
```

```
df.head()
```

```
import pandas as pd
final_df_images_features = pd.read_csv("final_df_image_features.csv")
```

```
final_df_images_features.head()
```

```
final_df_images_features["prev_72h_apparent_temperature"].iloc[0]
```

```
df["prev_72h_apparent_temperature"]
```

temporal fusion

```
temp.drop(columns=["prev_72h_weather"], inplace=True)
```

```
temp.info()
```

```
# ==========================
# 1. LOAD AND PREPROCESS DATA
# ==========================
# Load your

# Apply restructuring
restructured_df = restructure_dataframe(df)
# ==========================
# 2. PREPARE TIME-SERIES DATASET FOR TFT
# ==========================
# Define time-series dataset for Temporal Fusion Transformer
```

```python
# Correcting the TimeSeriesDataSet initialization



# Split dataset into training and validation sets
train_data, val_data = data.split_by_time()

train_dataloader = train_data.to_dataloader(train=True, batch_size=64)
val_dataloader = val_data.to_dataloader(train=False, batch_size=64)

# ==========================
# 3. TRAIN TEMPORAL FUSION TRANSFORMER MODEL
# ==========================
# Initialize Temporal Fusion Transformer model
tft = TemporalFusionTransformer.from_dataset(
    train_data,
    loss=QuantileLoss(),         # Quantile loss for probabilistic forecasting
    hidden_size=128,             # Hidden size of the model
    attention_head_size=4,       # Number of attention heads
    dropout=0.1,                 # Dropout rate to prevent overfitting
)

# Define optimizer and trainer settings
optimizer = torch.optim.Adam(tft.parameters(), lr=1e-3)

for epoch in range(30):  # Train for 30 epochs
    tft.train()
    epoch_loss = []
    for batch in train_dataloader:
        optimizer.zero_grad()
        loss = tft.training_step(batch)
        loss.backward()
        optimizer.step()
        epoch_loss.append(loss.item())

    print(f"Epoch {epoch + 1}: Loss = {np.mean(epoch_loss):.4f}")

# ==========================
# 4. EVALUATION METRICS AND PREDICTIONS
# ==========================
tft.eval()
val_predictions = []
val_targets = []
with torch.no_grad():
    for batch in val_dataloader:
        predictions = tft.predict(batch)
        targets = batch["target"].numpy()
        val_predictions.append(predictions.numpy())
        val_targets.append(targets)

val_predictions = np.concatenate(val_predictions)
val_targets = np.concatenate(val_targets)

roc_auc = roc_auc_score(val_targets, val_predictions)
balanced_acc = balanced_accuracy_score(val_targets, val_predictions.round())
```

```
print(f"Validation ROC AUC: {roc_auc:.4f}")
print(f"Validation Balanced Accuracy: {balanced_acc:.4f}")
```

```
temp.info()
```

```
# Restructure dataframe
def restructure_dataframe(df):
    rows = []

    # Identify all columns with the `prev_` prefix
    prev_columns = [col for col in df.columns if col.startswith('prev_')]

    for _, row in df.iterrows():
        # Extract static features
        static_features = {
            'event_id': row['event_id'],
            'event_type': row['event_type'],
            'begin_date_time': row['begin_date_time'],
            'image_features': np.array(row['image_features']),
            'extreme': row['extreme']
        }

        # Expand all `prev_` columns into individual rows
        num_timesteps = 10  # Assume all `prev_` columns have the same length (72 hours)

        for time_idx in range(num_timesteps):
            expanded_row = {**static_features, 'time_idx': time_idx}

            # Add values from each `prev_` column for the current timestep
            for col in prev_columns:
                expanded_row[col.replace('prev_', '')] = float(row[col][time_idx])

            rows.append(expanded_row)

    return pd.DataFrame(rows)

# Apply restructuring
restructured_df = restructure_dataframe(temp)

# Display restructured dataframe
print(restructured_df.head())
```

```
restructured_df.info()
```

```
import ast

def convert_to_array(val):
    if isinstance(val, str):
        return np.array(ast.literal_eval(val))
    return val

df['image_features'] = df['image_features'].apply(convert_to_array)
```

```

```

```
restructured_df.columns
```

```
def format_images(t):
    t = t.tolist()
    t = re.sub(r'[\n\s]+', ' ', t)
    t = t.replace("[", "").replace("]", "").replace("\n", "")
    return np.fromstring(t, sep=' ')
restructured_df["image_features"] = restructured_df["image_features"].apply(format_images)
```

```

```

```
df_model["combined_features"].iloc[0]
```

```
import pandas as pd
import numpy as np
import torch
from pytorch_forecasting import TemporalFusionTransformer
from pytorch_forecasting.data import TimeSeriesDataSet
from pytorch_forecasting.metrics import QuantileLoss
from sklearn.metrics import roc_auc_score, balanced_accuracy_score

df_model = restructured_df.copy()
# Combine temporal hourly features and image features into one array per row
hourly_features = [col for col in df_model.columns if "72h_" in col]
def combine_features(row):

    hourly_data = np.array([row[col] for col in hourly_features])

    image_data = row["image_features"]


    return np.concatenate([hourly_data, image_data])
df_model["combined_features"] = df_model.apply(combine_features, axis=1)
```

```
df_model.to_csv("df_model.csv", index=False)
```

```
df_model.head()
```

```
df_model["event_id"]=df_model["event_id"].apply(lambda x: str(x))
```

```
import pandas as pd
import torch
from pytorch_forecasting import TimeSeriesDataSet, TemporalFusionTransformer
from pytorch_forecasting.data import GroupNormalizer
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks import EarlyStopping

# Load your DataFrame (assuming it's already preprocessed)
df = df_model.copy() # Replace with your actual DataFrame file path

from pytorch_forecasting.data import TimeSeriesDataSet
# Fill missing timesteps for each group
df = df.groupby("event_type").apply(
```

```python
    lambda group: group.set_index("time_idx")
    .reindex(range(group["time_idx"].min(), group["time_idx"].max() + 1))
    .reset_index()
).reset_index(drop=True)

# Fill missing values with appropriate defaults (e.g., 0 or NaN)
df.fillna(0, inplace=True)  # Replace with appropriate default values for your dataset

# Define parameters for the TimeSeriesDataSet
max_encoder_length = 10  # Lookback period
max_prediction_length = 1  # Prediction horizon (extreme event classification)
batch_size = 64  # Batch size for training

# Create a TimeSeriesDataSet object
dataset = TimeSeriesDataSet(
    df,
    time_idx="time_idx",  # Column indicating the time step index
    target="extreme",  # Target column (binary classification: extreme or not)
    group_ids=["event_type"],  # Grouping by event type
    max_encoder_length=max_encoder_length,
    max_prediction_length=max_prediction_length,
    static_categoricals=["event_type"],  # Static categorical features
    time_varying_known_reals=[
        "72h_temperature_2m",
        "72h_relative_humidity_2m",
        "72h_dew_point_2m",
        "72h_apparent_temperature",
        "72h_precipitation",
        "72h_soil_temperature_28_to_100cm",
        "72h_soil_temperature_100_to_255cm",
        "72h_soil_moisture_0_to_7cm",
        "72h_soil_moisture_7_to_28cm",
        "72h_soil_moisture_28_to_100cm",
        "72h_soil_moisture_100_to_255cm",
    ],  # Features that are known at prediction time
    time_varying_unknown_reals=[],  # Add unknown features if applicable
    add_relative_time_idx=True,  # Add relative time index as a feature
    add_target_scales=True,  # Scale target variable
    add_encoder_length=True,  # Add encoder length as a feature
)
```

```python
# Inspect the time_idx column
time_idx_summary = {
    'min_time_idx': df['time_idx'].min(),
    'max_time_idx': df['time_idx'].max(),
    'unique_time_idx_count': df['time_idx'].nunique(),
    'time_idx_gaps': df['time_idx'].diff().value_counts().to_dict()
}
print(time_idx_summary)
```

```python
# Check for duplicates in time_idx within each event_type group
duplicates = df.groupby("event_type")["time_idx"].apply(lambda x: x.duplicated()).sum()
print(f"Number of duplicate time_idx values: {duplicates}")
```

```python
import openmeteo_requests
import requests_cache
import pandas as pd
from retry_requests import retry
import time
# Setup the Open-Meteo API client with caching and retry logic
cache_session = requests_cache.CachedSession('.cache', expire_after=-1)
retry_session = retry(cache_session, retries=5, backoff_factor=0.2)
openmeteo = openmeteo_requests.Client(session=retry_session)
# 30.703065642420704, -104.83971938166954
# Define a list of locations (with city, latitude, and longitude)
locations = [
    {"city": "New York City", "lat": 30.703065642420704, "lon": -104.83971938166954}
]

# API endpoint URL for the Open-Meteo archive
url = "https://archive-api.open-meteo.com/v1/archive"

# List to store dataframes for each location
dfs = []

for loc in locations:
    # Define API parameters for the location
    params = {
        "latitude": loc["lat"],
        "longitude": loc["lon"],
        "start_date": "1999-07-10",
        "end_date": "1999-07-10",
        "hourly": [
            "temperature_2m", "relative_humidity_2m", "dew_point_2m", "apparent_temperature",
            "precipitation", "rain", "snowfall", "snow_depth", "weather_code", "pressure_msl",
            "surface_pressure", "cloud_cover", "cloud_cover_low", "cloud_cover_mid", "cloud_co
            "et0_fao_evapotranspiration", "vapour_pressure_deficit", "wind_speed_10m", "wind_s
            "wind_direction_10m", "wind_direction_100m", "wind_gusts_10m", "soil_temperature_0
            "soil_temperature_7_to_28cm", "soil_temperature_28_to_100cm", "soil_temperature_10
            "soil_moisture_0_to_7cm", "soil_moisture_7_to_28cm", "soil_moisture_28_to_100cm",
            "soil_moisture_100_to_255cm"
        ]
    }

    # Call the API; responses is a list (here we use the first response)
    responses = openmeteo.weather_api(url, params=params)
    response = responses[0]

    # Process the hourly data; the order here must match the requested variables
    hourly = response.Hourly()
    hourly_temperature_2m             = hourly.Variables(0).ValuesAsNumpy()
    hourly_relative_humidity_2m       = hourly.Variables(1).ValuesAsNumpy()
    hourly_dew_point_2m               = hourly.Variables(2).ValuesAsNumpy()
    hourly_apparent_temperature       = hourly.Variables(3).ValuesAsNumpy()
    hourly_precipitation              = hourly.Variables(4).ValuesAsNumpy()
    hourly_rain                       = hourly.Variables(5).ValuesAsNumpy()
    hourly_snowfall                   = hourly.Variables(6).ValuesAsNumpy()
```

```python
    hourly_snow_depth                   = hourly.Variables(7).ValuesAsNumpy()
    hourly_weather_code                 = hourly.Variables(8).ValuesAsNumpy()
    hourly_pressure_msl                 = hourly.Variables(9).ValuesAsNumpy()
    hourly_surface_pressure             = hourly.Variables(10).ValuesAsNumpy()
    hourly_cloud_cover                  = hourly.Variables(11).ValuesAsNumpy()
    hourly_cloud_cover_low              = hourly.Variables(12).ValuesAsNumpy()
    hourly_cloud_cover_mid              = hourly.Variables(13).ValuesAsNumpy()
    hourly_cloud_cover_high             = hourly.Variables(14).ValuesAsNumpy()
    hourly_et0_fao_evapotranspiration   = hourly.Variables(15).ValuesAsNumpy()
    hourly_vapour_pressure_deficit      = hourly.Variables(16).ValuesAsNumpy()
    hourly_wind_speed_10m               = hourly.Variables(17).ValuesAsNumpy()
    hourly_wind_speed_100m              = hourly.Variables(18).ValuesAsNumpy()
    hourly_wind_direction_10m           = hourly.Variables(19).ValuesAsNumpy()
    hourly_wind_direction_100m          = hourly.Variables(20).ValuesAsNumpy()
    hourly_wind_gusts_10m               = hourly.Variables(21).ValuesAsNumpy()
    hourly_soil_temperature_0_to_7cm    = hourly.Variables(22).ValuesAsNumpy()
    hourly_soil_temperature_7_to_28cm   = hourly.Variables(23).ValuesAsNumpy()
    hourly_soil_temperature_28_to_100cm  = hourly.Variables(24).ValuesAsNumpy()
    hourly_soil_temperature_100_to_255cm = hourly.Variables(25).ValuesAsNumpy()
    hourly_soil_moisture_0_to_7cm       = hourly.Variables(26).ValuesAsNumpy()
    hourly_soil_moisture_7_to_28cm      = hourly.Variables(27).ValuesAsNumpy()
    hourly_soil_moisture_28_to_100cm    = hourly.Variables(28).ValuesAsNumpy()
    hourly_soil_moisture_100_to_255cm   = hourly.Variables(29).ValuesAsNumpy()

    # Create a date range from the hourly time information
    date_range = pd.date_range(
        start=pd.to_datetime(hourly.Time(), unit="s", utc=True),
        end=pd.to_datetime(hourly.TimeEnd(), unit="s", utc=True),
        freq=pd.Timedelta(seconds=hourly.Interval()),
        inclusive="left"
    )

    # Build a DataFrame for this location
    df = pd.DataFrame({
        "date": date_range,
        "temperature_2m": hourly_temperature_2m,
        "relative_humidity_2m": hourly_relative_humidity_2m,
        "dew_point_2m": hourly_dew_point_2m,
        "apparent_temperature": hourly_apparent_temperature,
        "precipitation": hourly_precipitation,
        "rain": hourly_rain,
        "snowfall": hourly_snowfall,
        "snow_depth": hourly_snow_depth,
        "weather_code": hourly_weather_code,
        "pressure_msl": hourly_pressure_msl,
        "surface_pressure": hourly_surface_pressure,
        "cloud_cover": hourly_cloud_cover,
        "cloud_cover_low": hourly_cloud_cover_low,
        "cloud_cover_mid": hourly_cloud_cover_mid,
        "cloud_cover_high": hourly_cloud_cover_high,
        "et0_fao_evapotranspiration": hourly_et0_fao_evapotranspiration,
        "vapour_pressure_deficit": hourly_vapour_pressure_deficit,
        "wind_speed_10m": hourly_wind_speed_10m,
        "wind_speed_100m": hourly_wind_speed_100m,
        "wind_direction_10m": hourly_wind_direction_10m,
        "wind_direction_100m": hourly_wind_direction_100m,
```

```
        "wind_gusts_10m": hourly_wind_gusts_10m,
        "soil_temperature_0_to_7cm": hourly_soil_temperature_0_to_7cm,
        "soil_temperature_7_to_28cm": hourly_soil_temperature_7_to_28cm,
        "soil_temperature_28_to_100cm": hourly_soil_temperature_28_to_100cm,
        "soil_temperature_100_to_255cm": hourly_soil_temperature_100_to_255cm,
        "soil_moisture_0_to_7cm": hourly_soil_moisture_0_to_7cm,
        "soil_moisture_7_to_28cm": hourly_soil_moisture_7_to_28cm,
        "soil_moisture_28_to_100cm": hourly_soil_moisture_28_to_100cm,
        "soil_moisture_100_to_255cm": hourly_soil_moisture_100_to_255cm
    })

    # Add location metadata
    df["city"] = loc["city"]
    df["latitude"] = loc["lat"]
    df["longitude"] = loc["lon"]

    # Append this location's DataFrame to the list
    dfs.append(df)
    time.sleep(30)

# Combine all location DataFrames into one DataFrame
combined_df = pd.concat(dfs, ignore_index=True)

# Save the combined DataFrame to a CSV file
combined_df.to_csv("us_weather_data.csv", index=False)

print("Data saved to us_weather_data.csv")
```

```
combined_df.info()
```

```
import requests
import urllib.parse

city = "Paris"
country = "France"
url = "https://nominatim.openstreetmap.org/ui/search.php.html?q=" + city + "+" + country +"&fo

response = requests.get(url).json()
print(response[0]["lat"])
print(response[0]["lon"])
```

```
combined_df.columns
```

```
import pandas as pd
import requests
import io
import datetime
from meteostat import Point, Daily

# ---------------------------
# 2. Download Wildfire Data
# ---------------------------
# Assume an open-source wildfire CSV is hosted online (for example, WildfireDB as in [1]).
# Replace the wildcard URL below with the actual link to the CSV file.
wildfire_url = 'https://wildfire-modeling.github.io/dataset/Historical_Wildfires.csv'
```

```
wf_response = requests.get(wildfire_url)
# Read CSV from text; if needed, adjust the delimiter or encoding.
print(wf_response)

# Convert date column to datetime and filter by time.
# Make sure the column names match those in your CSV (e.g., 'Date','latitude','longitude').
wildfire_df['Date'] = pd.to_datetime(wildfire_df['Date'], errors='coerce')
filtered_wf = wildfire_df[(wildfire_df['Date'] >= start_date) & (wildfire_df['Date'] <= end_da

# If you want to filter by a specific location (e.g., near a given lat/lon), you can add a bou
loc_lat = 34.0
loc_lon = -118.0
lat_tolerance = 1.0  # degrees
lon_tolerance = 1.0  # degrees
filtered_wf = filtered_wf[
    (wildfire_df['latitude'].between(loc_lat - lat_tolerance, loc_lat + lat_tolerance)) &
    (wildfire_df['longitude'].between(loc_lon - lon_tolerance, loc_lon + lon_tolerance))
]
print("Wildfires (filtered):")
print(filtered_wf.head())

# ---------------------------
# 3. Retrieve Heavy Rainfall Data using Meteostat
# ---------------------------
# Install meteostat package before running:
# pip install meteostat

# Define the location and time period
lat, lon = 34.0, -118.0  # example coordinates (e.g., near Los Angeles)
location = Point(lat, lon)
start = datetime.datetime(2020, 1, 1)
end = datetime.datetime(2020, 12, 31)

# Fetch daily weather data (includes precipitation 'prcp')
daily_data = Daily(location, start, end)
daily_data = daily_data.fetch()

# Optionally, filter for heavy rainfall events by setting a threshold (e.g., prcp > 20 mm)
threshold = 20.0  # in millimeters
heavy_rain = daily_data[daily_data['prcp'] >= threshold]
print("Heavy Rainfall Events (filtered):")
print(heavy_rain[['prcp']].head())
```

```
import requests
from geopy.geocoders import Nominatim

# Disable SSL verification for requests (TEMPORARY workaround)
requests.packages.urllib3.disable_warnings()

geolocator = Nominatim(user_agent="your_app_name", scheme="http")  # Use HTTP instead of HTTPS
location = geolocator.geocode("175 5th Avenue NYC")

print(location.address)
print((location.latitude, location.longitude))
```

```
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

```
  https://www.google.com/maps/embed/v1/MAP_MODE?key=AIzaSyCyX_pcJrqpv-SPoCrHOei1WaOmJC0Zuc4&pa
```

```
import googlemaps

gmaps = googlemaps.Client(key="AIzaSyCyX_pcJrqpv-SPoCrHOei1WaOmJC0Zuc4")
geocode_result = gmaps.geocode("175 5th Avenue NYC")

if geocode_result:
    location = geocode_result[0]['geometry']['location']
    print(location['lat'], location['lng'])
```

```
import ssl
import certifi
from geopy.geocoders import Nominatim

# Create an SSL context using the certifi certificate bundle
context = ssl.create_default_context(cafile=certifi.where())

# Pass the custom SSL context to Nominatim (supported in geopy 2.2+)
geolocator = Nominatim(user_agent="GetLoc", scheme="https", ssl_context=context)
location = geolocator.geocode("TATUM")

print(location.address)
print("Latitude =", location.latitude)
print("Longitude =", location.longitude)
```

```
import ssl
import certifi
import urllib.request
import pandas as pd

url = "https://www1.ncdc.noaa.gov/pub/data/swdi/stormevents/csvfiles/StormEvents_details-2021.

# Create an SSL context that uses certifi's CA bundle
ctx = ssl.create_default_context(cafile=certifi.where())

# Use urllib.request to open the URL with the SSL context
with urllib.request.urlopen(url, context=ctx) as response:
    df = pd.read_csv(response, compression='gzip')

print(df.head())
```

```
from owslib.wms import WebMapService
from datetime import datetime, timedelta
import os

# NASA GIBS WMS endpoint for EPSG:4326 (geographic coordinates)
wms_url = "https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi?"
wms = WebMapService(wms_url, version="1.1.1")

# Choose the MODIS Terra True Color layer (available daily)
```

```
layer = "MODIS_Terra_CorrectedReflectance_TrueColor"

# Define the area of interest: bounding box for New York City
# Format: [min_longitude, min_latitude, max_longitude, max_latitude]
bbox = [-74.25909, 40.477399, -73.700272, 40.917577]

# Set output image size (width, height)
img_size = (1600, 1200)

# Define the time range: last 5 years. We will download one image every 30 days.
end_date = datetime.utcnow().date()  # Today's date in UTC
start_date = end_date.replace(year=end_date.year - 5)

# Create an output directory to store the images
output_dir = "satellite_images"
os.makedirs(output_dir, exist_ok=True)

current_date = start_date
while current_date <= end_date:
    time_str = current_date.strftime("%Y-%m-%d")
    print(f"Downloading image for {time_str}...")
    try:
        # Request the image for the given date and area
        response = wms.getmap(
            layers=[layer],
            srs="EPSG:4326",
            bbox=bbox,
            size=img_size,
            time=time_str,
            format="image/png",
            transparent=True
        )
        # Save the image with the date in its filename
        filename = os.path.join(output_dir, f"satellite_{time_str}.png")
        with open(filename, "wb") as f:
            f.write(response.read())
        print(f"Saved {filename}")
    except Exception as e:
        print(f"Failed to download image for {time_str}: {e}")

    # Move forward by 30 days (approx. one month)
    current_date += timedelta(days=30)

print("Download process complete.")
```

```
%pip install owslib
```

```
from owslib.wmts import WebMapTileService
from PIL import Image
from io import BytesIO

# NASA GIBS WMTS endpoint for EPSG:4326
wmts_url = "https://gibs.earthdata.nasa.gov/wmts/epsg4326/best/wmts.cgi?"
wmts = WebMapTileService(wmts_url)
```

```
# Define parameters
layer = "MODIS_Terra_CorrectedReflectance_TrueColor"  # MODIS Terra True Color layer
date = "2025-02-01"  # Desired date
tile_matrix_set = "250m"  # Tile resolution (250 meters per pixel)

# Define the bounding box (latitude/longitude range)
lat_center, lon_center = 40.7128, -74.0060  # Example: New York City
bounding_box_size = 0.2  # Degrees around the center (creates a square area)
bbox = [
    lon_center - bounding_box_size,
    lat_center - bounding_box_size,
    lon_center + bounding_box_size,
    lat_center + bounding_box_size,
]

# Request the image from WMTS
tile_response = wmts.gettile(
    layer=layer,
    tilematrixset=tile_matrix_set,
    tilematrix="7",  # Zoom level; higher values give more detail
    row=20,  # Adjust based on your area of interest
    column=10,  # Adjust based on your area of interest
    format="image/jpeg",
    time=date,
)

# Save the image
filename = f"satellite_image_{date}.jpg"
with open(filename, "wb") as f:
    f.write(tile_response.read())

print(f"Image saved as {filename}")
```

```
from owslib.wms import WebMapService
from datetime import datetime, timedelta
import os

# NASA GIBS WMS endpoint for EPSG:4326 (geographic coordinates)
wms_url = "https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi?"
wms = WebMapService(wms_url, version="1.1.1")

# Choose the MODIS Terra True Color layer (available daily)
layer = "IMERG_Precipitation_Rate"

# Define the area of interest: bounding box for New York City
# Format: [min_longitude, min_latitude, max_longitude, max_latitude]
bbox = [-74.25909, 40.477399, -73.700272, 40.917577]
expansion_factor = 10  # Adjust this value to control how much larger you want the bounding bo

# Expand the bounding box
expanded_bbox = [
    bbox[0] - expansion_factor,  # minx
    bbox[1] - expansion_factor,  # miny
    bbox[2] + expansion_factor,  # maxx
    bbox[3] + expansion_factor   # maxy
]
```

```
print("Expanded Bounding Box:", expanded_bbox)

# Set output image size (width, height)
img_size = (800, 600)

# Define the time range: last 5 years. We will download one image every 30 days.
end_date = datetime.utcnow().date()  # Today's date in UTC
start_date = end_date.replace(year=end_date.year - 5)

# Create an output directory to store the images
output_dir = "satellite_images"
os.makedirs(output_dir, exist_ok=True)

current_date = start_date
while current_date <= end_date:
    time_str = current_date.strftime("%Y-%m-%d")
    print(f"Downloading image for {time_str}...")
    try:
        # Request the image for the given date and area
        response = wms.getmap(
            layers=[layer],
            srs="EPSG:4326",
            bbox=bbox,
            size=img_size,
            time=time_str,
            format="image/png",
            transparent=True
        )
        # Save the image with the date in its filename
        filename = os.path.join(output_dir, f"satellite_{time_str}.png")
        with open(filename, "wb") as f:
            f.write(response.read())
        print(f"Saved {filename}")
    except Exception as e:
        print(f"Failed to download image for {time_str}: {e}")

    # Move forward by 30 days (approx. one month)
    current_date += timedelta(days=30)

print("Download process complete.")
```

```
import xml.etree.ElementTree as ET

# Read the XML content from a file
file_path = "wms.xml"  # Replace with your actual file path
with open(file_path, 'r', encoding='utf-8') as file:
    xml_content = file.read()

# Parse the XML content
root = ET.fromstring(xml_content)

# Define a recursive function to extract Name and Title from layers
def extract_layer_info(layer):
    name = layer.find('Name')
    title = layer.find('Title')
```

```
    if name is not None and title is not None:
        print(f"Name: {name.text}, Title: {title.text}")

    # Recursively check for nested layers
    for sublayer in layer.findall('Layer'):
        extract_layer_info(sublayer)

# Start extracting from the root Layer element
for layer in root.findall(".//Layer"):
    extract_layer_info(layer)
```

Data Collection

1. Extreme events

```
# wildfires

import pandas as pd

# Load the wildfire data from a CSV file
wildfire_data = pd.read_csv("/Users/etloaner/Documents/ASU/Capstone project/Wildfire events/WF
wildfire_data.head()
```

```
wildfire_data.info()
```

```
wildfire_data["IncidentSize"].value_counts()
```

```
wildfire_data["FinalAcres"].value_counts()
```

```
wildfire_data["FireCause"].value_counts()
```

```
wildfire_data["FireBehaviorGeneral"].value_counts()
```

```
wildfire_data["FireCauseGeneral"].value_counts()
```

```
wildfire_data["FireCauseSpecific"].value_counts()
```

```
wildfire_data["IncidentComplexityLevel"].value_counts()
```

```
wildfire_data["IncidentTypeCategory"].value_counts()
```

```
new_wf_df = wildfire_data[["OBJECTID", "IncidentSize", "EstimatedCostToDate", "FinalAcres", "F
```

```
new_wf_df.info()
```

```
new_wf_df.columns
```

```
final_wf_df = new_wf_df[new_wf_df["FireCause"] == "Natural"].drop(columns=["EstimatedCostToDat
```

```
new_wf_df = new_wf_df[['OBJECTID', 'IncidentSize', 'FireCause', 'InitialLatitude', 'InitialLon
```

```
final_wf_df.info()
```

```
final_wf_df.dropna(inplace=True)
```

```
final_wf_df.to_csv("wildfire_data_filtered.csv", index=False)
```

```
final_wf_df["FireCause"].value_counts()
```

```
import os
import gzip
import pandas as pd

# Define folder path where the files are stored
folder_path = "/Users/etloaner/Documents/ASU/Capstone project/Storm event dataset"

# Function to read compressed CSV files
def read_gz_csv(file_path):
    with gzip.open(file_path, "rt") as f:
        return pd.read_csv(f, low_memory=False)

# Lists to store data
details_list = []
locations_list = []
fatalities_list = []

# Read each file and append to corresponding list
for file in os.listdir(folder_path):
    file_path = os.path.join(folder_path, file)

    if "details" in file:
        df = read_gz_csv(file_path)
        df.columns = df.columns.str.strip().str.lower()  # Standardizing column names
        details_list.append(df)
    elif "locations" in file:
        df = read_gz_csv(file_path)
        df.columns = df.columns.str.strip().str.lower()
        locations_list.append(df)
    elif "fatalities" in file:
        df = read_gz_csv(file_path)
        df.columns = df.columns.str.strip().str.lower()
        fatalities_list.append(df)

# Concatenate all dataframes (append)
details_df = pd.concat(details_list, ignore_index=True) if details_list else pd.DataFrame()
locations_df = pd.concat(locations_list, ignore_index=True) if locations_list else pd.DataFran
fatalities_df = pd.concat(fatalities_list, ignore_index=True) if fatalities_list else pd.DataF

# Print column names for debugging
print("Details Columns:", details_df.columns)
print("Locations Columns:", locations_df.columns)
print("Fatalities Columns:", fatalities_df.columns)
```

```python
# Ensure 'event_id' exists in all DataFrames before merging
if "event_id" not in details_df.columns:
    raise KeyError("event_id missing from details dataset")
if "event_id" not in locations_df.columns:
    raise KeyError("event_id missing from locations dataset")
if "event_id" not in fatalities_df.columns:
    raise KeyError("event_id missing from fatalities dataset")

# Merge data using event_id
merged_df = details_df.merge(locations_df, on="event_id", how="left") \
                      .merge(fatalities_df, on="event_id", how="left")

# Save final dataframe to CSV
output_file = "final_storm_data.csv"
merged_df.to_csv(output_file, index=False)

print(f"Final consolidated dataset saved as {output_file}")
```

```python
import pandas as pd
storm_df = pd.read_csv("/Users/etloaner/Documents/ASU/Capstone project/final_storm_data.csv")
```

```python
storm_df.info()
```

```python
storm_df["event_type"].value_counts().index
```

```python
storm_df[storm_df["event_type"] == "Wildfire"].info()
```

```python
import pandas as pd
import numpy as np

# Function to convert damage strings (e.g., "10.00K", "10.00M") to a numeric value
def convert_damage(damage_str):
    try:
        if pd.isnull(damage_str):
            return 0.0
        damage_str = str(damage_str).strip()
        if damage_str.endswith('K'):
            return float(damage_str[:-1]) * 1e3
        elif damage_str.endswith('M'):
            return float(damage_str[:-1]) * 1e6
        else:
            return float(damage_str)
    except Exception:
        return 0.0

# Function to categorize an event as extreme (1) or non-extreme (0)
def categorize_extreme(row):
    # Get the event type and ensure no extra spaces
    event_type = str(row.get('event_type', '')).strip()

    # Convert damages to numeric values
    damage_property = convert_damage(row.get('damage_property', '0.00K'))
    damage_crops = convert_damage(row.get('damage_crops', '0.00K'))
    total_damage = damage_property + damage_crops
```

```python
    # Sum injuries and deaths from direct and indirect counts
    total_injuries = (row.get('injuries_direct', 0) or 0) + (row.get('injuries_indirect', 0) o
    total_deaths = (row.get('deaths_direct', 0) or 0) + (row.get('deaths_indirect', 0) or 0)

    # Set default thresholds (adjust these based on your domain knowledge)
    damage_threshold = 100000       # Example: $100K damage threshold
    injury_threshold = 20           # Example: 20 or more injuries
    death_threshold = 5             # Example: 5 or more deaths
    wind_speed_threshold = 50       # Wind speed in knots for extreme wind events
    hail_size_threshold = 1.5       # Hail size in inches considered extreme

    # Overall impact: if damage, injuries, or deaths exceed thresholds, mark as extreme.
    if total_damage >= damage_threshold or total_injuries >= injury_threshold or total_deaths
        return 1

    # Specific rules per event type
    if event_type == 'Tornado':
        # Use the Enhanced Fujita Scale: EF2 or higher is considered extreme.
        tor_scale = str(row.get('tor_f_scale', 'EF0')).strip()
        try:
            scale_value = int(tor_scale.replace('EF', ''))
        except Exception:
            scale_value = 0
        if scale_value >= 2:
            return 1

    if event_type in ['Thunderstorm Wind', 'High Wind', 'Strong Wind', 'Marine Thunderstorm Wi
        # For wind events, use the magnitude (assumed to be in knots)
        try:
            magnitude = float(row.get('magnitude', 0))
        except Exception:
            magnitude = 0
        if magnitude >= wind_speed_threshold:
            return 1

    if event_type == 'Hail':
        # For hail events, use the magnitude (assumed to be in inches)
        try:
            magnitude = float(row.get('magnitude', 0))
        except Exception:
            magnitude = 0
        if magnitude >= hail_size_threshold:
            return 1

    if event_type in ['Flash Flood', 'Flood', 'Coastal Flood']:
        # Flood events may be extreme if they cause high damage (already checked above)
        if total_damage >= damage_threshold:
            return 1

    if event_type in ['Heavy Rain', 'Excessive Heat', 'Heat']:
        # These events may be extreme if they result in significant injuries or damage.
        if total_injuries >= injury_threshold or total_damage >= damage_threshold:
            return 1

    if event_type in ['Winter Storm', 'Winter Weather', 'Blizzard']:
        # Winter events are flagged based on impact
```

```
        if total_damage >= damage_threshold or total_injuries >= injury_threshold:
            return 1

    # Additional event-specific rules can be added here

    # If none of the conditions are met, mark as non-extreme.
    return 0

# Load your consolidated dataset (ensure that column names are standardized, e.g., lower-case
df = pd.read_csv("final_storm_data.csv")
df.columns = df.columns.str.strip().str.lower()  # e.g., converting "Event_Type" to "event_typ

# Apply the categorization function to each row in the DataFrame
df['extreme'] = df.apply(categorize_extreme, axis=1)

# Save the updated DataFrame with the extreme flag to a new CSV file
output_file = "final_storm_data_with_extreme_flag.csv"
df.to_csv(output_file, index=False)

print(f"Categorization complete. Saved final file as '{output_file}'.")
```

```
storm_df["damage_property"].value_counts()
```

```
import pandas as pd
final_extreme_event_dataset = pd.read_csv("/Users/etloaner/Documents/ASU/Capstone project/fina
final_extreme_event_dataset.info()
```

```
final_extreme_event_dataset["extreme"].value_counts()
```

```
final_extreme_event_dataset["event_type"].value_counts()
```

```
final_extreme_event_dataset["year"].value_counts().index
```

```
filtered_df = final_extreme_event_dataset[final_extreme_event_dataset["year"].isin([2012, 2013
```

```
filtered_df.drop(columns=["episode_id_x","event_id","state_fips","cz_type","cz_fips","cz_name"
```

```
filtered_df.info()
```

```
filtered_df.columns
```

```
filtered_df = filtered_df[["begin_day", "begin_time", "end_day", "end_time", "event_type", "be
```

```
filtered_df.dropna(inplace=True)
```

```
filtered_df.info()
```

```
filtered_df["begin_date_time"].value_counts()
```

```
filtered_df.drop(columns=["begin_day","begin_time","end_day","end_time"], inplace=True)
```

```
import pandas as pd
from datetime import datetime
import pytz

df = filtered_df.copy()
# Mapping of timezone abbreviations to UTC offsets
timezone_offsets = {
    'CST-6': -6,
    'EST-5': -5,
    'MST-7': -7,
    'PST-8': -8,
    'AST-4': -4,
    'HST-10': -10,
    'AKST-9': -9,
    'SST-11': -11,
    'GST10': 10
}

# Function to convert local time to UTC
def convert_to_utc(local_time_str, timezone):
    local_time = datetime.strptime(local_time_str, '%d-%b-%y %H:%M:%S')
    offset = timezone_offsets.get(timezone, 0)
    local_time = local_time.replace(tzinfo=pytz.FixedOffset(offset * 60))
    utc_time = local_time.astimezone(pytz.utc)
    return utc_time

# Apply the function to the begin_date_time and end_date_time columns
df['begin_date_time_utc'] = df.apply(lambda row: convert_to_utc(row['begin_date_time'], row['c
df['end_date_time_utc'] = df.apply(lambda row: convert_to_utc(row['end_date_time'], row['cz_ti

# Split the datetime objects into separate date and time columns
df['begin_date_utc'] = df['begin_date_time_utc'].dt.date
df['begin_time_utc'] = df['begin_date_time_utc'].dt.time
df['end_date_utc'] = df['end_date_time_utc'].dt.date
df['end_time_utc'] = df['end_date_time_utc'].dt.time

# Drop the intermediate UTC datetime columns
df.drop(columns=['begin_date_time_utc', 'end_date_time_utc'], inplace=True)
```

```
df.to_csv("temp.csv", index=False)
```

```
import pandas as pd
from owslib.wms import WebMapService
from datetime import datetime, timedelta
import os
from tqdm import tqdm

# NASA GIBS WMS endpoint for EPSG:4326 (geographic coordinates)
wms_url = "https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi?"
wms = WebMapService(wms_url, version="1.1.1")

# Choose the MODIS Terra True Color layer (daily imagery)
layer = "MODIS_Terra_CorrectedReflectance_TrueColor"

# Set output image size (width, height)
```

```python
img_size = (800, 600)

# Create an output directory for images and metadata
output_dir = "satellite_images"
os.makedirs(output_dir, exist_ok=True)

# List to collect metadata for each downloaded image
metadata_records = []

# Define a margin (in degrees) to expand the event bounding box (if event is a point)
margin = 0.1

# Process each event row
for idx, row in tqdm(df.iterrows(), total=len(df), desc="Processing events"):
    # Compute the bounding box: [min_lon, min_lat, max_lon, max_lat]
    min_lon = min(row["begin_lon"], row["end_lon"]) - margin
    max_lon = max(row["begin_lon"], row["end_lon"]) + margin
    min_lat = min(row["begin_lat"], row["end_lat"]) - margin
    max_lat = max(row["begin_lat"], row["end_lat"]) + margin
    bbox = [min_lon, min_lat, max_lon, max_lat]

    # Combine the UTC date and time to get the event datetime, then subtract a delta (e.g., 3
    event_dt_str = f"{row['begin_date_utc']}T{row['begin_time_utc']}Z"
    event_dt = datetime.strptime(event_dt_str, "%Y-%m-%dT%H:%M:%SZ")
    # Use an image timestamp 3 hours before the event (adjust as needed)
    image_dt = event_dt - timedelta(hours=3)

    # Format the time string in ISO8601 (required by the WMS service)
    time_str = image_dt.strftime("%Y-%m-%dT%H:%M:%SZ")

    # Create a safe version of the event type (remove spaces)
    safe_event_type = row["event_type"].replace(" ", "_")

    # Construct a filename that incorporates event id, type, and timestamp
    filename = os.path.join(output_dir, f"event{idx}_{safe_event_type}_{image_dt.strftime('%Y%

    print(f"Downloading image for event {idx} ({row['event_type']}) at {time_str} with bbox {b
    try:
        response = wms.getmap(
            layers=[layer],
            srs="EPSG:4326",
            bbox=bbox,
            size=img_size,
            time=time_str,
            format="image/png",
            transparent=True
        )
        with open(filename, "wb") as f:
            f.write(response.read())
        print(f"Saved image to {filename}")

        # Append metadata record
        metadata_records.append({
            "event_id": idx,
            "event_type": row["event_type"],
            "image_filename": os.path.basename(filename),
```

```
            "download_time": datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%SZ"),
            "image_time": time_str,
            "bbox": bbox,
            "event_time": event_dt_str
        })
    except Exception as e:
        print(f"Failed to download image for event {idx}: {e}")

# Save metadata to a CSV file
metadata_df = pd.DataFrame(metadata_records)
metadata_csv = os.path.join(output_dir, "satellite_images_metadata.csv")
metadata_df.to_csv(metadata_csv, index=False)
print(f"Saved metadata to {metadata_csv}")
```

```
from sentinelsat import SentinelAPI, read_geojson, geojson_to_wkt

# Connect to Copernicus Open Access Hub (free Sentinel data)
api = SentinelAPI('guest', 'guest', 'https://scihub.copernicus.eu/dhus')


# Define your area of interest (AOI) using a GeoJSON file or WKT string
geojson_path = "sample.geojson"  # Replace with your GeoJSON file path
footprint = geojson_to_wkt(read_geojson(geojson_path))

# Define the time range in the correct format (YYYYMMDD)
start_date = '20250201'  # Correct format for sentinelsat
end_date = '20250217'    # Correct format for sentinelsat
cloud_cover = (0, 20)    # Only images with 0-20% cloud cover

# Search for Sentinel-2 products
products = api.query(
    footprint,
    date=(start_date, end_date),  # Ensure dates are in YYYYMMDD format
    platformname='Sentinel-2',
    cloudcoverpercentage=cloud_cover
)

# Print available products
print(f"Found {len(products)} products.")
for product_id, product_info in products.items():
    print(f"ID: {product_id}, Title: {product_info['title']}")

# Download the first product (if available)
if products:
    product_id = list(products.keys())[0]
    print(f"Downloading product: {products[product_id]['title']}")
    api.download(product_id, directory_path='./satellite_data')
else:
    print("No suitable products found.")
```

```
df["event_type"].value_counts()
```

```
import math
import requests
from PIL import Image
```

```python
from io import BytesIO
import pandas as pd

# Function to calculate tile indices for a given latitude, longitude, and zoom level
def lat_lon_to_tile_indices(lat, lon, zoom):
    n = 2 ** zoom
    x_tile = int((lon + 180.0) / 360.0 * n)
    y_tile = int((1.0 - math.log(math.tan(math.radians(lat)) + 1 / math.cos(math.radians(lat))
    return x_tile, y_tile

# NASA GIBS WMTS endpoint
wmts_url = "https://gibs.earthdata.nasa.gov/wmts/epsg4326/best/wmts.cgi?"

# Define parameters
layer = "MODIS_Terra_CorrectedReflectance_TrueColor"
tile_matrix_set = "250m"  # Tile resolution (250 meters per pixel)
zoom_level = 6  # Adjust based on desired detail

# Assuming 'df' is your DataFrame loaded with your event data
for index, event in df[df["event_type"] == "Heavy Rain"].iterrows():
    # Use the provided columns for coordinates:
    # 'begin_lat' and 'begin_lon' indicate the start point of the event.
    # 'end_lat' and 'end_lon' indicate the end point of the event.
    begin_lat = event["begin_lat"]
    begin_lon = event["begin_lon"]
    end_lat = event["end_lat"]
    end_lon = event["end_lon"]

    # Option 1: Use the beginning coordinate directly
    # x_tile, y_tile = lat_lon_to_tile_indices(begin_lat, begin_lon, zoom_level)

    # Option 2: Compute the midpoint between the begin and end coordinates for a representativ
    mid_lat = (begin_lat + end_lat) / 2
    mid_lon = (begin_lon + end_lon) / 2
    x_tile, y_tile = lat_lon_to_tile_indices(mid_lat, mid_lon, zoom_level)

    # Use the begin_date_utc column for the time parameter (format: YYYY-MM-DD)
    date = event["begin_date_utc"]

    # Construct WMTS request URL
    tile_url = (
        f"{wmts_url}SERVICE=WMTS&REQUEST=GetTile&VERSION=1.0.0"
        f"&LAYER={layer}&STYLE=default&FORMAT=image/jpeg"
        f"&TILEMATRIXSET={tile_matrix_set}&TILEMATRIX={zoom_level}"
        f"&TILEROW={y_tile}&TILECOL={x_tile}&TIME={date}"
    )

    # Download tile image
    output_dir = "satellite_images"
    os.makedirs(output_dir, exist_ok=True)
    response = requests.get(tile_url)
    if response.status_code == 200:
        img = Image.open(BytesIO(response.content))
        # Save the image with a filename that includes the midpoint coordinates and date
        filename = os.path.join(output_dir, f"satellite_{mid_lat}_{mid_lon}_{date}.jpg")
        img.save(filename)
```

```
        print(f"Image saved: {filename}")
    else:
        print(f"Failed to download image for coordinates ({mid_lat}, {mid_lon}) on {date}: {re
```

```
df
```

```
import math
import requests
from PIL import Image
from io import BytesIO
import os
from datetime import datetime, timedelta

def lat_lon_to_tile_indices_epsg4326(lat, lon, zoom):
    minx, maxx = -180, 180
    miny, maxy = -90, 90
    matrix_width = 2 ** (zoom + 1)
    matrix_height = 2 ** zoom
    tile_width = (maxx - minx) / matrix_width
    tile_height = (maxy - miny) / matrix_height
    tile_col = int((lon - minx) / tile_width)
    tile_row = int((maxy - lat) / tile_height)
    return tile_col, tile_row


# Coordinates for Phoenix (update comment if necessary)
lat = 32.1649
lon = -110.8706
zoom_level = 8

# Create a directory to save images (naming it to reflect the target location)
output_dir = "phoenix_tiles_may_2023"
os.makedirs(output_dir, exist_ok=True)

# Loop through each day of May 2023
start_date = datetime.strptime("2023-05-01", "%Y-%m-%d")
end_date = datetime.strptime("2023-05-31", "%Y-%m-%d")
current_date = start_date

while current_date <= end_date:
    date_str = current_date.strftime("%Y-%m-%d")

    # Calculate EPSG:4326 tile indices
    tile_col, tile_row = lat_lon_to_tile_indices_epsg4326(lat, lon, zoom_level)
    print("Tile indices (EPSG:4326): Column =", tile_col, "Row =", tile_row)

    # Construct the RESTful WMTS URL (using the proper tile matrix set identifier)
    base_url = "https://gibs.earthdata.nasa.gov/wmts/epsg4326/best/"
    layer = "MODIS_Terra_CorrectedReflectance_TrueColor"
    tile_matrix_set = "250m"  # Updated from "250m" to full identifier
    tile_url = (
        f"{base_url}{layer}/default/{date_str}/{tile_matrix_set}/{zoom_level}/{tile_row}/{tile
    )

    print("Requesting tile URL:")
```

```
    print(tile_url)

    # Download the tile image
    response = requests.get(tile_url)
    if response.status_code == 200:
        img = Image.open(BytesIO(response.content))
        filename = os.path.join(output_dir, f"phoenix_tile_{date_str}.jpg")
        img.save(filename)
        print(f"Tile image saved as {filename}")
    else:
        print("Failed to download image:", response.status_code)

    # Move to the next day
    current_date += timedelta(days=1)
```

```
from sentinelsat import SentinelAPI, read_geojson, geojson_to_wkt
from datetime import date

# Replace with your CDSE credentials
username = 'nmistry6@asu.edu'
password = 'Naman@7415963'

# Connect to the API
api = SentinelAPI(username, password, 'https://scihub.copernicus.eu/dhus')

# Define the area of interest (Phoenix, Arizona)
footprint = geojson_to_wkt(read_geojson('sample.geojson'))

# Define the date range for May 2023
start_date = date(2023, 5, 1)
end_date = date(2023, 5, 31)

# Search for Sentinel-2 products
products = api.query(footprint,
                     date=(start_date, end_date),
                     platformname='Sentinel-2',
                     cloudcoverpercentage=(0, 30))

# Convert to Pandas DataFrame
products_df = api.to_dataframe(products)

# Display found products
print(products_df[['title', 'beginposition', 'cloudcoverpercentage']])
```

```
import math
import requests
from PIL import Image
from io import BytesIO

def lat_lon_to_tile(lat, lon, zoom):
    """Convert latitude and longitude to tile coordinates at a given zoom level."""
    lat_rad = math.radians(lat)
    n = 2.0 ** zoom
    x_tile = int((lon + 180.0) / 360.0 * n)
    y_tile = int((1.0 - math.log(math.tan(lat_rad) + 1 / math.cos(lat_rad)) / math.pi) / 2.0 *
```

```python
    return x_tile, y_tile

def download_satellite_image(lat, lon, zoom, output_file):
    """Download satellite imagery for a specific location and save it as an image."""
    x_tile, y_tile = lat_lon_to_tile(lat, lon, zoom)

    # Example: Esri Satellite Imagery URL template
    url_template = "https://services.arcgisonline.com/arcgis/rest/services/World_Imagery/MapSe
    url = url_template.format(z=zoom, x=x_tile, y=y_tile)

    response = requests.get(url)
    if response.status_code == 200:
        # Save the image
        image = Image.open(BytesIO(response.content))
        image.save(output_file)
        print(f"Image saved as {output_file}")
    else:
        print(f"Failed to download image: HTTP {response.status_code}")

# Parameters

latitude = 32.1649  # Latitude (e.g., New York City)
longitude = -110.8706  # Longitude
zoom_level = 16  # Zoom level (higher values mean higher resolution)
output_filename = "satellite_image.png"

# Download the satellite image
download_satellite_image(latitude, longitude, zoom_level, output_filename)
```

```python
import math
import requests
from PIL import Image
from io import BytesIO

def lat_lon_to_tile_nasa(lat, lon, level):
    """
    Converts latitude and longitude to the corresponding NASA GIBS WMTS tile indices.

    For NASA GIBS in EPSG:4326, the tile matrix is defined as:
      - Number of columns: 2^(level+1)
      - Number of rows:    2^(level)

    The origin is at (-180, 90) (top-left corner).
    """
    cols = 2 ** (level + 1)
    rows = 2 ** level
    tile_width = 360.0 / cols
    tile_height = 180.0 / rows
    tile_col = int((lon + 180) / tile_width)
    tile_row = int((90 - lat) / tile_height)
    return tile_col, tile_row

def download_tile(layer, date_str, resolution, level, tile_row, tile_col):
    """
    Downloads a single tile image from NASA GIBS WMTS endpoint.
```

```
    Constructs the URL:
      https://gibs.earthdata.nasa.gov/wmts/epsg4326/best/{layer}/default/{date_str}/{resolutio
    Returns a PIL Image if successful, otherwise None.
    """
    base_url = "https://gibs.earthdata.nasa.gov/wmts/epsg4326/best"
    url = f"{base_url}/{layer}/default/{date_str}/{resolution}/{level}/{tile_row}/{tile_col}.j
    response = requests.get(url)
    if response.status_code == 200:
        return Image.open(BytesIO(response.content))
    else:
        print(f"Failed to download tile (row {tile_row}, col {tile_col}): HTTP {response.statu
        return None

def download_and_stitch_tiles(lat, lon, date_str, level=3, block_size=3,
                              layer="MODIS_Terra_Cloud_Mask_TrueColor",
                              resolution="250m"):
    """
    Downloads and stitches a block (block_size x block_size) of NASA GIBS WMTS tiles centered

    Note: This code does not apply any additional cloud masking; it returns the imagery as pro

    Parameters:
      lat, lon     : Center point coordinates.
      date_str     : Date in 'YYYY-MM-DD' format (time-of-day is not provided).
      level        : Tile matrix level (higher value means a smaller area per tile).
      block_size   : Number of tiles per side in the resulting mosaic (should be odd to center
      layer        : The GIBS imagery layer.
      resolution   : Resolution of the imagery (e.g., "250m").

    Returns:
      A stitched PIL Image.
    """
    center_tile_col, center_tile_row = lat_lon_to_tile_nasa(lat, lon, level)

    # Ensure block_size is odd so that the center tile corresponds to the provided lat/lon.
    if block_size % 2 == 0:
        block_size += 1

    offset = block_size // 2
    tiles = []

    for r in range(center_tile_row - offset, center_tile_row + offset + 1):
        row_tiles = []
        for c in range(center_tile_col - offset, center_tile_col + offset + 1):
            print(f"Downloading tile at row {r}, col {c}...")
            tile_img = download_tile(layer, date_str, resolution, level, r, c)
            if tile_img is None:
                # If a tile fails to download, create a blank placeholder.
                tile_img = Image.new("RGB", (256, 256), (0, 0, 0))
            row_tiles.append(tile_img)
        tiles.append(row_tiles)

    tile_width, tile_height = tiles[0][0].size
    total_width = tile_width * block_size
    total_height = tile_height * block_size
    stitched_image = Image.new("RGB", (total_width, total_height))
```

```
    for i, row_tiles in enumerate(tiles):
        for j, tile in enumerate(row_tiles):
            stitched_image.paste(tile, (j * tile_width, i * tile_height))

    return stitched_image

# User-defined parameters:
latitude = 40.7128         # Example: New York City latitude
longitude = -74.0060       # Example: New York City longitude
date_str = "2024-07-15"    # Date in 'YYYY-MM-DD' format (time-of-day not provided)
zoom_level = 8             # Tile matrix level (adjust for desired detail)
block_size = 5             # Mosaic block size (5x5 tiles)

# Download and stitch the tiles, then save to a file.
final_image = download_and_stitch_tiles(latitude, longitude, date_str,
                                        level=zoom_level, block_size=block_size)
final_image.save("weather_satellite_image.jpg")
print("Downloaded and saved 'weather_satellite_image.jpg'.")
```

```
import lxml.etree as xmltree
# Construct capability URL.
wmsUrl = 'https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi?\
SERVICE=WMS&REQUEST=GetCapabilities'

# Request WMS capabilities.
response = requests.get(wmsUrl)

# Display capabilities XML in original format. Tag and content in one line.
WmsXml = xmltree.fromstring(response.content)
# print(xmltree.tostring(WmsXml, pretty_print = True, encoding = str))
```

```
# Currently total layers are 1081.
import xml.etree.ElementTree as xmlet
# Convert capability response to XML tree.
WmtsTree = xmlet.fromstring(response.content)

alllayer = []
layerNumber = 0

# Parse capability XML tree.
for child in WmtsTree.iter():
    for layer in child.findall("./{http://www.opengis.net/wmts/1.0}Layer"):
         if '{http://www.opengis.net/wmts/1.0}Layer' == layer.tag:
            f=layer.find("{http://www.opengis.net/ows/1.1}Identifier")
            if f is not None:
                alllayer.append(f.text)
                layerNumber += 1

# Print the first five and last five layers.
print('Number of layers: ', layerNumber)
for one in sorted(alllayer)[:5]:
    print(one)
print('...')
```

```
for one in sorted(alllayer)[-5:]:
    print(one)
```

```
import requests
from PIL import Image
from io import BytesIO

def download_wms_image(lat, lon, date_str, layer, width=512, height=512, extent=0.1):
    """
    Downloads a map image from NASA GIBS using WMS.

    Parameters:
      lat       : Latitude of the center point.
      lon       : Longitude of the center point.
      date_str  : Date in 'YYYY-MM-DD' format.
      layer     : The GIBS imagery layer.
      width     : Width of the output image in pixels.
      height    : Height of the output image in pixels.
      extent    : Half of the desired bounding box size in degrees.
                  The bounding box extends 'extent' degrees in each direction from the center.

    Returns:
      A PIL Image with the requested satellite view.
    """
    # Define a bounding box around the center point.
    # WMS (version 1.1.1) expects bbox as: min_lon, min_lat, max_lon, max_lat.
    lon_min = lon - extent
    lat_min = lat - extent
    lon_max = lon + extent
    lat_max = lat + extent
    bbox = f"{lon_min},{lat_min},{lon_max},{lat_max}"

    # NASA GIBS WMS endpoint. Using version 1.1.1 ensures the SRS parameters and bbox format c
    wms_url = "https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi"

    # Define WMS parameters.
    params = {
        "service": "WMS",
        "version": "1.1.1",
        "request": "GetMap",
        "layers": layer,
        "styles": "",
        "format": "image/jpeg",
        "transparent": "true",
        "srs": "EPSG:4326",
        "bbox": bbox,
        "width": width,
        "height": height,
        "time": date_str  # Specify the date for time-enabled layers.
    }

    response = requests.get(wms_url, params=params)
    if response.status_code == 200:
        return Image.open(BytesIO(response.content))
    else:
        print(f"Failed to download WMS image: HTTP {response.status_code}")
```

```
        return None

# User-defined parameters.
latitude = 42.9371        # Example: New York City latitude.
longitude = -75.6107       # Example: New York City longitude.
date_str = "2022-07-15"    # Date in 'YYYY-MM-DD' format.
layer = "MODIS_Terra_Cloud_Top_Temp_Day"  # Example layer; adjust if needed.

# Download the WMS image.
final_image = download_wms_image(latitude, longitude, date_str, layer, width=2014, height=2014
if final_image:
    final_image.save("weather_satellite_wms.jpg")
    print("Downloaded and saved 'weather_satellite_wms.jpg'.")
```

```
import requests
from xml.etree import ElementTree

def get_gibs_layers():
    # NASA GIBS WMS GetCapabilities URL
    url = "https://gibs.earthdata.nasa.gov/wms/epsg4326/best/wms.cgi?service=WMS&request=GetCa
    response = requests.get(url)
    if response.status_code == 200:
        # Parse the XML response
        tree = ElementTree.fromstring(response.content)
        layers = []
        for layer in tree.findall(".//{http://www.opengis.net/wms}Layer"):
            title = layer.find("{http://www.opengis.net/wms}Title").text if layer.find("{http:
            name = layer.find("{http://www.opengis.net/wms}Name").text if layer.find("{http://
            layers.append((name, title))
        return layers
    else:
        print(f"Failed to fetch layers: HTTP {response.status_code}")
        return []

# Fetch and print available GIBS layers
layers = get_gibs_layers()
for name, title in layers:
    print(f"Layer Name: {name}, Title: {title}")
```

```
import pandas as pd

df = pd.read_csv(r"C:\Personal\Capstone project\Capstone project\final_storm_data.csv")
```

```
t = df[(df["event_type"] == "Thunderstorm Wind") & (df["magnitude"] <=30)].iloc[5].to_dict()
t
```

```
from datetime import datetime, timedelta
import pandas as pd
import openmeteo_requests
import requests_cache
```

```python
from retry_requests import retry

# Initialize caching and retry mechanisms
cache_session = requests_cache.CachedSession('.cache', expire_after=-1)
retry_session = retry(cache_session, retries=5, backoff_factor=0.2)
openmeteo = openmeteo_requests.Client(session=retry_session)

# User-specified parameters
latitude = t["latitude"]
longitude = t["longitude"]
end_date_time = t["end_date_time"]
cz_timezone = 'CST-6'  # Central Standard Time (UTC-6)

# Convert end_date_time to UTC and calculate start_date_time (72 hours before)
end_date_utc = datetime.strptime(end_date_time, '%d-%b-%y %H:%M:%S') - timedelta(hours=8) + ti
start_date_utc = end_date_utc - timedelta(hours=72)

# Prepare parameters for the API request
params = {
    "latitude": latitude,
    "longitude": longitude,
    "start_date": start_date_utc.strftime('%Y-%m-%d'),
    "end_date": end_date_utc.strftime('%Y-%m-%d'),
    "hourly": [
        "temperature_2m",
        "relative_humidity_2m",
        "dew_point_2m",
        "apparent_temperature",
        "precipitation",
        "rain",
        "snowfall",
        "snow_depth",
        "weather_code",
        "pressure_msl",
        "surface_pressure",
        "cloud_cover",
        "wind_speed_10m",
        "wind_direction_10m",
        "wind_gusts_10m",
        "soil_temperature_0_to_7cm",
        "soil_moisture_0_to_7cm"
    ]
}

# Fetch data from Open-Meteo API
responses = openmeteo.weather_api(
    "https://archive-api.open-meteo.com/v1/archive",
    params=params
)
response = responses[0]

# Process hourly data from the response
hourly = response.Hourly()
data = {
    "date": pd.date_range(
        start=pd.to_datetime(hourly.Time(), unit="s", utc=True),
```

```
        end=pd.to_datetime(hourly.TimeEnd(), unit="s", utc=True),
        freq=pd.Timedelta(seconds=hourly.Interval()),
        inclusive="left"
    ),
    "latitude": latitude,
    "longitude": longitude
}

# Add all weather variables to the data dictionary
for idx, var in enumerate(params["hourly"]):
    data[var] = hourly.Variables(idx).ValuesAsNumpy()

# Convert the data dictionary into a DataFrame for further analysis or saving
weather_data_df = pd.DataFrame(data)

# Display or save the DataFrame as needed
# print(weather_data_df)
```

```
weather_data_df["wind_gusts_10m"].max()
```

```
weather_data_df.columns
```

```
# create a line chart
weather_data_df["wind_speed_10m"].plot(title="Wind Gusts (10m)", xlabel="Date", ylabel="Wind G
```

```
# create a line chart
weather_data_df["wind_direction_10m"].plot(title="Wind Gusts (10m)", xlabel="Date", ylabel="Wi
```

```
# create a line chart
weather_data_df["wind_gusts_10m"].plot(title="Wind Gusts (10m)", xlabel="Date", ylabel="Wind G
```

```
from datetime import datetime, timedelta, timezone
import pandas as pd
import matplotlib.pyplot as plt
import openmeteo_requests
import requests_cache
from retry_requests import retry
import numpy as np

# ---------------------------
# Setup caching and Open-Meteo client
# ---------------------------
cache_session = requests_cache.CachedSession('.cache', expire_after=-1)
retry_session = retry(cache_session, retries=5, backoff_factor=0.2)
openmeteo = openmeteo_requests.Client(session=retry_session)

# ---------------------------
# Load NOAA Dataset
# ---------------------------
# Replace 'noaa_data.csv' with the path to your NOAA dataset CSV file.
# df = pd.read_csv("noaa_data.csv")

# ---------------------------
# Filter for Thunderstorm Wind events
```

```
# ---------------------------
extreme = df[(df["event_type"] == "Thunderstorm Wind") & (df["magnitude"] >=70)].iloc[0].to_di

non_extreme = df[(df["event_type"] == "Thunderstorm Wind") & (df["magnitude"] <=5)].iloc[5].to

# ---------------------------
# Helper function to parse timezone from NOAA field
# ---------------------------
def parse_timezone(tz_str):
    """
    Parse a timezone string from NOAA dataset (e.g., "PST-8", "CST-6")
    and return a datetime.timezone object.
    """
    # Check if tz_str contains '-' or '+'
    if '-' in tz_str:
        parts = tz_str.split('-')
        offset = -abs(int(parts[1]))
    elif '+' in tz_str:
        parts = tz_str.split('+')
        offset = abs(int(parts[1]))
    else:
        offset = 0  # default to UTC if not specified
    return timezone(timedelta(hours=offset))

# ---------------------------
# Define a function to fetch weather data for an event
# ---------------------------
def fetch_weather_data(event):
    """
    Given an event dictionary, fetch weather data from Open-Meteo for the 72 hours prior
    to the event's end_date_time.
    The function converts the local event time (using cz_timezone) to UTC.
    """
    latitude = event["latitude"]
    longitude = event["longitude"]
    end_date_time_str = event["end_date_time"]  # e.g., "04-FEB-24 17:09:00"
    tz_str = event.get("cz_timezone", "UTC")     # default to UTC if not provided

    event_tz = parse_timezone(tz_str)

    # Parse the end_date_time in local time and assign the proper timezone
    end_local = datetime.strptime(end_date_time_str, '%d-%b-%y %H:%M:%S').replace(tzinfo=event

    # Convert the local time to UTC - *** REMOVED THE INCORRECT +timedelta(hours=5) ***
    end_date_utc = end_local.astimezone(timezone.utc)

    # Define the start of the window (72 hours prior to the END time)
    start_date_utc = end_date_utc - timedelta(hours=72)

    print(f"Event Local End Time: {end_local}")
    print(f"Fetching Data Window (UTC): {start_date_utc} to {end_date_utc}")

    # Prepare parameters for the API request - Fetch one extra day on end_date
    # because the API end_date is inclusive of the day but we want hourly data up to the speci
    # Fetching data up to the *day* of end_date_utc ensures we get the hour of the event.
    params = {
```

```
        "latitude": latitude,
        "longitude": longitude,
        "start_date": start_date_utc.strftime('%Y-%m-%d'),
        # Fetch data up to and including the day the event ends
        "end_date": end_date_utc.strftime('%Y-%m-%d'),
        "hourly": [
            "temperature_2m", "relative_humidity_2m", "dew_point_2m",
            "apparent_temperature", "precipitation", "rain", "snowfall",
            "snow_depth", "weather_code", "pressure_msl", "surface_pressure",
            "cloud_cover", "wind_speed_10m", "wind_direction_10m",
            "wind_gusts_10m", "soil_temperature_0_to_7cm", "soil_moisture_0_to_7cm"
        ],
        "wind_speed_unit": "kn" # Requesting knots
    }

    responses = openmeteo.weather_api("https://archive-api.open-meteo.com/v1/archive", params=
    response = responses[0]

    hourly = response.Hourly()

    times = pd.date_range(
        start=pd.to_datetime(hourly.Time(), unit="s", utc=True),
        end=pd.to_datetime(hourly.TimeEnd(), unit="s", utc=True),
        freq=pd.Timedelta(seconds=hourly.Interval()),
        inclusive="left"
    )

    data = { "date": times }
    # Map response variables directly using the order in params['hourly']
    hourly_variables = params["hourly"]
    for i, var_name in enumerate(hourly_variables):
         data[var_name] = hourly.Variables(i).ValuesAsNumpy()

    weather_df = pd.DataFrame(data)

    # Filter the dataframe to the precise 72-hour window ending at the event time
    # because the API might return full days.
    weather_df = weather_df[(weather_df['date'] > start_date_utc) & (weather_df['date'] <= end
    # Add lat/lon back after filtering if needed, or keep them from the start if preferred
    weather_df["latitude"] = latitude
    weather_df["longitude"] = longitude

    return weather_df

# ---------------------------
# Fetch weather data for extreme and non-extreme events
# ---------------------------
weather_extreme = fetch_weather_data(extreme)
weather_non_extreme = fetch_weather_data(non_extreme)

# ---------------------------
# Plotting the time-series wind speed data for comparison
# ---------------------------
plt.figure(figsize=(14, 6))

# Plot extreme event wind speed
```

```
plt.subplot(1, 2, 1)
plt.plot(weather_extreme["date"], weather_extreme["wind_gusts_10m"], label="Wind Speed")
plt.title("Extreme Thunderstorm Wind Event")
plt.xlabel("Date (UTC)")
plt.ylabel("Wind Speed (m/s)")
plt.xticks(rotation=45)
plt.grid(True)
plt.legend()

# Plot non-extreme event wind speed
plt.subplot(1, 2, 2)
plt.plot(weather_non_extreme["date"], weather_non_extreme["wind_gusts_10m"], label="Wind Speed
plt.title("Non-Extreme Thunderstorm Wind Event")
plt.xlabel("Date (UTC)")
plt.ylabel("Wind Speed (m/s)")
plt.xticks(rotation=45)
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()

# ---------------------------
# Additional Analysis
# ---------------------------
# You can add further analysis and visualization (e.g., temperature, humidity) to understand c
```

```
import pandas as pd
from datetime import datetime, timedelta, timezone
import matplotlib.pyplot as plt
import openmeteo_requests
import requests_cache
from retry_requests import retry
import numpy as np

# ---------------------------
# Setup caching and Open-Meteo client (Keep as is)
# ---------------------------
cache_session = requests_cache.CachedSession('.cache', expire_after=-1)
retry_session = retry(cache_session, retries=5, backoff_factor=0.2)
openmeteo = openmeteo_requests.Client(session=retry_session)

# ---------------------------
# Load NOAA Dataset (Keep as is)
# ---------------------------
# Replace 'noaa_data.csv' with the path to your NOAA dataset CSV file.
# try:
#     df = pd.read_csv("noaa_data.csv", low_memory=False) # Added low_memory=False for potenti
# except FileNotFoundError:
#     print("Error: noaa_data.csv not found. Please ensure the file exists.")
#     exit()
# except Exception as e:
#     print(f"Error reading CSV: {e}")
#     exit()

# ---------------------------
```

```
# Filter for Thunderstorm Wind events (Keep as is, ensure events exist)
# ---------------------------
try:
    extreme_events = df[(df["event_type"] == "Thunderstorm Wind") & (df["magnitude"] >= 70)]
    if extreme_events.empty:
        print("Warning: No 'Thunderstorm Wind' events found with magnitude >= 70.")
        # Handle this case - maybe exit, or use a lower threshold for demonstration
        # For now, let's try a lower threshold if >= 70 fails
        extreme_events = df[(df["event_type"] == "Thunderstorm Wind") & (df["magnitude"] >= 50
        if extreme_events.empty:
             print("Error: No suitable extreme events found even at lower threshold.")
             exit()
        print("Using magnitude >= 50 for extreme event example.")

    extreme = extreme_events.iloc[0].to_dict()

    non_extreme_events = df[(df["event_type"] == "Thunderstorm Wind") & (df["magnitude"] <= 15
    if non_extreme_events.empty:
        print("Error: No suitable non-extreme 'Thunderstorm Wind' events found (magnitude 1-15
        exit()
    # Ensure we don't pick the exact same event if indices are reset or data is small
    non_extreme_index = min(5, len(non_extreme_events) - 1) # Pick index 5, or the last one if
    non_extreme = non_extreme_events.iloc[non_extreme_index].to_dict()

except IndexError:
    print("Error: Could not find enough matching events in the DataFrame. Check filtering crit
    exit()
except KeyError as e:
    print(f"Error: Column {e} not found in CSV. Check column names.")
    exit()


# ---------------------------
# Helper function to parse timezone (Keep as is)
# ---------------------------
def parse_timezone(tz_str):
    """
    Parse a timezone string from NOAA dataset (e.g., "PST-8", "CST-6")
    and return a datetime.timezone object. Handles non-standard entries more robustly.
    """
    if not isinstance(tz_str, str):
        print(f"Warning: Invalid timezone format '{tz_str}'. Defaulting to UTC.")
        return timezone.utc # Default to UTC if format is unexpected

    # Try to extract offset directly if it's just like "-5" or "+10"
    try:
        offset_hours = int(tz_str)
        # Basic sanity check for realistic offsets
        if -14 <= offset_hours <= 14:
             return timezone(timedelta(hours=offset_hours))
    except ValueError:
        pass # Continue to other parsing methods if direct int conversion fails

    # Handle formats like "XXX-offset" or "XXX+offset"
    if '-' in tz_str:
        parts = tz_str.split('-')
```

```
        try:
            # Get the last part assuming it's the offset
            offset = -abs(int(parts[-1]))
            if -14 <= offset <= 14:
                 return timezone(timedelta(hours=offset))
        except (ValueError, IndexError):
             pass # Ignore if parsing fails
    elif '+' in tz_str:
         parts = tz_str.split('+')
         try:
            offset = abs(int(parts[-1]))
            if -14 <= offset <= 14:
                return timezone(timedelta(hours=offset))
         except (ValueError, IndexError):
            pass # Ignore if parsing fails

    # If all else fails or format is unrecognized (like just "CST")
    print(f"Warning: Could not parse timezone offset from '{tz_str}'. Defaulting to UTC.")
    return timezone.utc # Default to UTC

# ---------------------------
# Define an ENHANCED function to fetch weather data
# ---------------------------
def fetch_enhanced_weather_data(event):
    """
    Given an event dictionary, fetch weather data from Open-Meteo's ERA5
    reanalysis for the 72 hours prior to the event's end_date_time.
    Includes surface, instability, shear potential, and moisture variables.
    Converts local event time (using cz_timezone) to UTC.
    """
    try:
        latitude = event["latitude"]
        longitude = event["longitude"]
        end_date_time_str = event["end_date_time"]
        tz_str = event.get("cz_timezone", "UTC")

        event_tz = parse_timezone(tz_str)

        # Parse the end_date_time in local time and assign the proper timezone
        # Use errors='coerce' for robustness against potential format issues if needed, but tr
        end_local = datetime.strptime(end_date_time_str, '%d-%b-%y %H:%M:%S').replace(tzinfo=e

        # Convert the local time to UTC
        end_date_utc = end_local.astimezone(timezone.utc)
        # Define the start of the window (72 hours prior to the END of the event)
        start_date_utc = end_date_utc - timedelta(hours=72)

        print(f"\nFetching data for event ending:")
        print(f"  End Local: {end_local.strftime('%Y-%m-%d %H:%M:%S %Z%z')}")
        print(f"  End UTC:   {end_date_utc.strftime('%Y-%m-%d %H:%M:%S %Z')}")
        print(f"  Start UTC: {start_date_utc.strftime('%Y-%m-%d %H:%M:%S %Z')}")
        print(f"  Lat/Lon:   {latitude}, {longitude}")

        # Prepare parameters for the API request - ADDING MANY MORE VARIABLES
        params = {
            "latitude": latitude,
```

```
        "longitude": longitude,
        "start_date": start_date_utc.strftime('%Y-%m-%d'),
        "end_date": end_date_utc.strftime('%Y-%m-%d'),
        "hourly": [
            # Basic Surface Vars
            "temperature_2m", "relative_humidity_2m", "dew_point_2m",
            "apparent_temperature", "pressure_msl", "surface_pressure",
            "precipitation", "rain", "snowfall", "weather_code", "cloud_cover",
            # Surface Winds
            "wind_speed_10m", "wind_direction_10m", "wind_gusts_10m",
            # --- ENHANCED VARIABLES ---
            # Instability Indices (Crucial for thunderstorms)
            "cape",             # Convective Available Potential Energy (J/kg) - Higher me
            "lifted_index",     # Lifted Index (K) - Negative values indicate instability
            # Moisture at different levels (Understanding vertical moisture profile)
            "relative_humidity_1000hPa", "relative_humidity_850hPa",
            "relative_humidity_700hPa", "relative_humidity_500hPa",
            # Temperature at different levels (For calculating lapse rates -> instability)
            "temperature_1000hPa", "temperature_850hPa",
            "temperature_700hPa", "temperature_500hPa",
            # Winds at different levels (For calculating shear -> storm organization/sever
            "wind_speed_850hPa", "wind_direction_850hPa",
            "wind_speed_700hPa", "wind_direction_700hPa",
            "wind_speed_500hPa", "wind_direction_500hPa",
             # Geopotential Height (Shows troughs/ridges which influence weather patterns)
            "geopotential_height_850hPa", "geopotential_height_500hPa"
            # Optional: Add more levels (e.g., 925hPa, 300hPa) or other vars if needed
        ],
        "wind_speed_unit": "kn", # Use knots to be closer to NOAA magnitude units
         # Specify ERA5 explicitly for potentially better data consistency
        "models": "era5"
    }

    # Fetch weather data from Open-Meteo API
    print(f"  Requesting {len(params['hourly'])} variables...")
    responses = openmeteo.weather_api("https://archive-api.open-meteo.com/v1/archive", par
    response = responses[0]
    print(f"  Data received successfully.")

    # Process hourly data
    hourly = response.Hourly()

    # Create time range
    times = pd.date_range(
        start=pd.to_datetime(hourly.Time(), unit="s", utc=True),
        end=pd.to_datetime(hourly.TimeEnd(), unit="s", utc=True),
        freq=pd.Timedelta(seconds=hourly.Interval()),
        inclusive="left"
    )

    data = {
        "date": times,
        "latitude": latitude, # Keep original requested lat/lon
        "longitude": longitude,
        "api_latitude": response.Latitude(),
        "api_longitude": response.Longitude()
```

```
        }

        hourly_variables = response.Hourly() # Get the hourly object once
        requested_var_names = params["hourly"]
        num_vars_api_returned = 0 # Initialize count

        # It's safer to check if VariablesLength method exists and use it
        if hasattr(hourly_variables, 'VariablesLength'):
             num_vars_api_returned = hourly_variables.VariablesLength()
             print(f"  API reported {num_vars_api_returned} variables returned.")
        else:
             # Fallback or warning if VariablesLength is not available (unlikely for recent ve
             print("  Warning: Cannot determine exact number of variables returned by API via
             # We might have to rely on the requested length, which is less robust
             num_vars_api_returned = len(requested_var_names)


        # Iterate based on the ORDER of requested variables.
        # The API usually returns them in the same order.
        for idx, var_name in enumerate(requested_var_names):
            # Check if the index is within the bounds of what the API actually returned
            if idx < num_vars_api_returned:
                try:
                    # Access the variable data by its index 'idx'
                    variable = hourly_variables.Variables(idx) # Pass the index 'idx'
                    values = variable.ValuesAsNumpy()

                    print(f"    Processing: {var_name} (index {idx}, {len(values)} values)")

                    # Check for length consistency
                    if len(times) != len(values):
                        print(f"      *** Length mismatch for {var_name}! Expected {len(times)
                        # Ensure the NaN array has the correct length expected by the 'times'
                        data[var_name] = np.full(len(times), np.nan)
                    else:
                        data[var_name] = values

                except TypeError as te:
                    # Specifically catch the error you encountered
                    print(f"      *** TypeError accessing variable '{var_name}' at index {idx}
                    print(f"      *** This likely indicates an issue with the library version
                    data[var_name] = np.full(len(times), np.nan) # Fill with NaNs
                except IndexError:
                    # This shouldn't happen with the idx < num_vars_api_returned check, but fo
                    print(f"      *** Error: Index {idx} out of bounds unexpectedly for variab
                    data[var_name] = np.full(len(times), np.nan)
                except Exception as e:
                    # Catch any other unexpected errors during processing
                    print(f"      *** Unexpected error processing variable '{var_name}' at inc
                    data[var_name] = np.full(len(times), np.nan)
            else:
                # If API returned fewer variables than requested
                print(f"    Skipping: {var_name} (index {idx}) - Variable not found in API res
                data[var_name] = np.full(len(times), np.nan) # Fill with NaNs

        weather_df = pd.DataFrame(data)
```

```
        # Set date as index AFTER DataFrame creation and filling data
        weather_df = weather_df.set_index('date')
        return weather_df

    except Exception as e:
        print(f"Error fetching or processing data for event: {e}")
        # Print event details that caused the error
        print("Event details:", event)
        return pd.DataFrame() # Return empty DataFrame on error

# ---------------------------
# Fetch ENHANCED weather data for events
# ---------------------------
print("Fetching data for EXTREME event...")
weather_extreme_enhanced = fetch_enhanced_weather_data(extreme)

print("\nFetching data for NON-EXTREME event...")
weather_non_extreme_enhanced = fetch_enhanced_weather_data(non_extreme)

# ---------------------------
# Basic Data Checks
# ---------------------------
print("\n--- Data Check ---")
print(f"Extreme event data shape: {weather_extreme_enhanced.shape}")
print(f"Non-extreme event data shape: {weather_non_extreme_enhanced.shape}")

# Check if data was fetched successfully
if weather_extreme_enhanced.empty or weather_non_extreme_enhanced.empty:
    print("\nError: Data fetching failed for one or both events. Cannot proceed with plotting.
else:
    print("\nColumns available:", weather_extreme_enhanced.columns.tolist())
    print("\nSample extreme data head:")
    print(weather_extreme_enhanced.head())

    # ---------------------------
    # Plotting Examples (Include some new variables)
    # ---------------------------
    print("\nGenerating plots...")
    plt.style.use('seaborn-v0_8-whitegrid') # Use a nice style

    fig, axes = plt.subplots(3, 2, figsize=(16, 15), sharex=True) # 3 rows, 2 columns

    # --- Row 1: Wind Gusts ---
    axes[0, 0].plot(weather_extreme_enhanced.index, weather_extreme_enhanced["wind_gusts_10m"]
    axes[0, 0].set_title(f"Extreme Event (NOAA Mag: {extreme.get('magnitude', 'N/A')} kn)")
    axes[0, 0].set_ylabel("Wind Gusts (kn)")
    axes[0, 0].legend()
    axes[0, 0].grid(True)

    axes[0, 1].plot(weather_non_extreme_enhanced.index, weather_non_extreme_enhanced["wind_gus
    axes[0, 1].set_title(f"Non-Extreme Event (NOAA Mag: {non_extreme.get('magnitude', 'N/A')}
    axes[0, 1].set_ylabel("Wind Gusts (kn)")
    axes[0, 1].legend()
    axes[0, 1].grid(True)

    # --- Row 2: CAPE (Instability) ---
```

```
    axes[1, 0].plot(weather_extreme_enhanced.index, weather_extreme_enhanced["cape"], label="C
    axes[1, 0].set_ylabel("CAPE (J/kg)")
    axes[1, 0].legend()
    axes[1, 0].grid(True)

    axes[1, 1].plot(weather_non_extreme_enhanced.index, weather_non_extreme_enhanced["cape"],
    axes[1, 1].set_ylabel("CAPE (J/kg)")
    axes[1, 1].legend()
    axes[1, 1].grid(True)

    # --- Row 3: Dew Point (Surface Moisture) ---
    axes[2, 0].plot(weather_extreme_enhanced.index, weather_extreme_enhanced["dew_point_2m"],
    axes[2, 0].set_ylabel("Dew Point (°C)")
    axes[2, 0].set_xlabel("Date (UTC)")
    axes[2, 0].legend()
    axes[2, 0].grid(True)

    axes[2, 1].plot(weather_non_extreme_enhanced.index, weather_non_extreme_enhanced["dew_poir
    axes[2, 1].set_ylabel("Dew Point (°C)")
    axes[2, 1].set_xlabel("Date (UTC)")
    axes[2, 1].legend()
    axes[2, 1].grid(True)

    # Improve layout and display
    plt.xticks(rotation=45, ha='right') # Rotate x-axis labels on the last row
    fig.suptitle("Comparison of Meteorological Conditions (72 hours prior)", fontsize=16)
    plt.tight_layout(rect=[0, 0.03, 1, 0.95]) # Adjust layout to prevent title overlap
    plt.show()

    print("\n--- Next Steps ---")
    print("1. Analyze the new variables: Look for significant differences in CAPE, Lifted Inde
    print("2. Calculate Derived Variables:")
    print("   - Lapse Rates: Calculate temperature difference between levels (e.g., T_500hPa -
    print("   - Wind Shear: Calculate the vector difference between winds at different levels
    print("3. Feature Engineering: Create rolling statistics (mean, max, std dev, rate of char
    print("4. Re-train Model: Use these enhanced and derived features as input to your ML/DL m
    print("5. Consider Problem Reframing: If predicting the exact NOAA magnitude is still harc

# Example of calculating a derived variable (Lapse Rate Estimate)
if not weather_extreme_enhanced.empty and 'temperature_850hPa' in weather_extreme_enhanced.col
     # Note: This is a simplified lapse rate. Accurate calculation needs geopotential height.
     # Higher positive values indicate steeper (more unstable) lapse rates.
     weather_extreme_enhanced['lapse_rate_850_500'] = weather_extreme_enhanced['temperature_85
     print("\nAdded 'lapse_rate_850_500' estimate to extreme data.")
     # You would do the same for non_extreme data and then plot or use this feature.
```

```
from datetime import datetime
import matplotlib.pyplot as plt
from meteostat import Point, Hourly
import pandas as pd
import numpy as np
def fetch_weather_data(latitude, longitude, start_date, end_date):
    """
    Fetch historical weather data for a specific location and date range.
    """
    # Create a Point for the location
```

```
    location = Point(latitude, longitude)

    # Set the time period
    start = datetime.strptime(start_date, '%Y-%m-%d')
    end = datetime.strptime(end_date, '%Y-%m-%d')

    # Fetch hourly data
    data = Hourly(location, start, end)
    data = data.fetch()

    return data
# Coordinates for the location (e.g., Chandler, Arizona)
latitude = 33.3062
longitude = -111.8413

# Define date ranges for analysis
start_date_extreme = '2024-02-01'
end_date_extreme = '2024-02-28'

start_date_non_extreme = '2024-03-01'
end_date_non_extreme = '2024-03-31'

# Fetch weather data
weather_extreme = fetch_weather_data(latitude, longitude, start_date_extreme, end_date_extreme
weather_non_extreme = fetch_weather_data(latitude, longitude, start_date_non_extreme, end_date
# Calculate the 90th percentile for wind gusts
threshold_extreme = np.percentile(weather_extreme['wsg10'], 90)
threshold_non_extreme = np.percentile(weather_non_extreme['wsg10'], 90)

# Categorize events
weather_extreme['event_type'] = np.where(weather_extreme['wsg10'] >= threshold_extreme, 'Extre
weather_non_extreme['event_type'] = np.where(weather_non_extreme['wsg10'] >= threshold_non_ext
plt.figure(figsize=(14, 6))

# Plot extreme event wind speed
plt.subplot(1, 2, 1)
plt.plot(weather_extreme.index, weather_extreme['wsg10'], label='Wind Speed', color='blue')
plt.axhline(y=threshold_extreme, color='red', linestyle='--', label='90th Percentile Threshold
plt.fill_between(weather_extreme.index, weather_extreme['wsg10'], threshold_extreme, where=(we
plt.title("Extreme Thunderstorm Wind Event")
plt.xlabel("Date")
plt.ylabel("Wind Speed (m/s)")
plt.xticks(rotation=45)
plt.grid(True)
plt.legend()

# Plot non-extreme event wind speed
plt.subplot(1, 2, 2)
plt.plot(weather_non_extreme.index, weather_non_extreme['wsg10'], label='Wind Speed', color='c
plt.axhline(y=threshold_non_extreme, color='green', linestyle='--', label='90th Percentile Thr
plt.fill_between(weather_non_extreme.index, weather_non_extreme['wsg10'], threshold_non_extrem
plt.title("Non-Extreme Thunderstorm Wind Event")
plt.xlabel("Date")
plt.ylabel("Wind Speed (m/s)")
plt.xticks(rotation=45)
plt.grid(True)
```

```
plt.legend()

plt.tight_layout()
plt.show()
```

```
for i in df.isna().sum().sort_values(ascending=False).items():
    if i[1]/df.shape[0] > 0.4:
        print(i[0], i[1])
        df.drop(columns=[i[0]], inplace=True)
```

```
945887/df.shape[0]
```

```
df = df[df["year"] > 2007]
```

```
%pip install meteostat
```

```
  from meteostat import Stations, Hourly
  from datetime import datetime

  # Set time period
  start = datetime(2024, 2, 1)
  end = datetime(2024, 2, 28)

  # Get nearby weather stations
  stations = Stations()
  stations = stations.nearby(37.51, -122.27)  # Example coordinates
  station = stations.fetch(1)

  # Fetch hourly data
  data = Hourly(station, start, end)
  data = data.fetch()

  # Display data
  print(data)
```

```
import pandas as pd



import pandas as pd
import numpy as np

# Function to convert damage strings (e.g., "10.00K", "10.00M") to a numeric value
def convert_damage(damage_str):
    try:
        if pd.isnull(damage_str):
            return 0.0
        damage_str = str(damage_str).strip()
        if damage_str.endswith('K'):
            return float(damage_str[:-1]) * 1e3
        elif damage_str.endswith('M'):
            return float(damage_str[:-1]) * 1e6
        else:
            return float(damage_str)
```

```
    except Exception:
        return 0.0

# Function to categorize an event as extreme (1) or non-extreme (0)
def categorize_extreme(row):
    # Get the event type and ensure no extra spaces
    event_type = str(row.get('event_type', '')).strip()

    # Convert damages to numeric values
    damage_property = convert_damage(row.get('damage_property', '0.00K'))
    damage_crops = convert_damage(row.get('damage_crops', '0.00K'))
    total_damage = damage_property + damage_crops

    # Sum injuries and deaths from direct and indirect counts
    total_injuries = (row.get('injuries_direct', 0) or 0) + (row.get('injuries_indirect', 0) o
    total_deaths = (row.get('deaths_direct', 0) or 0) + (row.get('deaths_indirect', 0) or 0)

    # Set default thresholds (adjust these based on your domain knowledge)
    damage_threshold = 100000       # Example: $100K damage threshold
    injury_threshold = 20           # Example: 20 or more injuries
    death_threshold = 5             # Example: 5 or more deaths
    wind_speed_threshold = 50       # Wind speed in knots for extreme wind events
    hail_size_threshold = 1.5       # Hail size in inches considered extreme

    # Overall impact: if damage, injuries, or deaths exceed thresholds, mark as extreme.
    if total_damage >= damage_threshold or total_injuries >= injury_threshold or total_deaths
        return 1

    # Specific rules per event type
    if event_type == 'Tornado':
        # Use the Enhanced Fujita Scale: EF2 or higher is considered extreme.
        tor_scale = str(row.get('tor_f_scale', 'EF0')).strip()
        try:
            scale_value = int(tor_scale.replace('EF', ''))
        except Exception:
            scale_value = 0
        if scale_value >= 2:
            return 1

    if event_type in ['Thunderstorm Wind', 'High Wind', 'Strong Wind', 'Marine Thunderstorm Wi
        # For wind events, use the magnitude (assumed to be in knots)
        try:
            magnitude = float(row.get('magnitude', 0))
        except Exception:
            magnitude = 0
        if magnitude >= wind_speed_threshold:
            return 1

    if event_type == 'Hail':
        # For hail events, use the magnitude (assumed to be in inches)
        try:
            magnitude = float(row.get('magnitude', 0))
        except Exception:
            magnitude = 0
        if magnitude >= hail_size_threshold:
            return 1
```

```
    if event_type in ['Flash Flood', 'Flood', 'Coastal Flood']:
        # Flood events may be extreme if they cause high damage (already checked above)
        if total_damage >= damage_threshold:
            return 1

    if event_type in ['Heavy Rain', 'Excessive Heat', 'Heat']:
        # These events may be extreme if they result in significant injuries or damage.
        if total_injuries >= injury_threshold or total_damage >= damage_threshold:
            return 1

    if event_type in ['Winter Storm', 'Winter Weather', 'Blizzard']:
        # Winter events are flagged based on impact
        if total_damage >= damage_threshold or total_injuries >= injury_threshold:
            return 1

    # Additional event-specific rules can be added here

    # If none of the conditions are met, mark as non-extreme.
    return 0


df.columns = df.columns.str.strip().str.lower()  # e.g., converting "Event_Type" to "event_typ

# Apply the categorization function to each row in the DataFrame
df['extreme'] = df.apply(categorize_extreme, axis=1)

#
```

```
extreme = df[(df["event_type"] == "Flash Flood") & (df["begin_yearmonth"] == 201508) & (df["ex
```

```
non_extreme = df[(df["event_type"] == "Flash Flood") & (df["begin_yearmonth"] == 201508) & (df
```

```
import pandas as pd
from datetime import datetime
import pytz

# Mapping of timezone abbreviations to UTC offsets
timezone_offsets = {
    'CST-6': -6,
    'EST-5': -5,
    'MST-7': -7,
    'PST-8': -8,
    'AST-4': -4,
    'HST-10': -10,
    'AKST-9': -9,
    'SST-11': -11,
    'GST10': 10
}

# Function to convert local time to UTC
def convert_to_utc(local_time_str, timezone):
    local_time = datetime.strptime(local_time_str, '%d-%b-%y %H:%M:%S')
    offset = timezone_offsets.get(timezone, 0)
    local_time = local_time.replace(tzinfo=pytz.FixedOffset(offset * 60))
```

```
    utc_time = local_time.astimezone(pytz.utc)
    return utc_time

# Apply the function to the begin_date_time and end_date_time columns
non_extreme['begin_date_time_utc'] = non_extreme.apply(lambda row: convert_to_utc(row['begin_d
non_extreme['end_date_time_utc'] = non_extreme.apply(lambda row: convert_to_utc(row['end_date_

# Split the datetime objects into separate date and time columns
non_extreme['begin_date_utc'] = non_extreme['begin_date_time_utc'].dt.date
non_extreme['begin_time_utc'] = non_extreme['begin_date_time_utc'].dt.time
non_extreme['end_date_utc'] = non_extreme['end_date_time_utc'].dt.date
non_extreme['end_time_utc'] = non_extreme['end_date_time_utc'].dt.time

# Drop the intermediate UTC datetime columns
non_extreme.drop(columns=['begin_date_time_utc', 'end_date_time_utc'], inplace=True)
```

```
row.index
```

```
import openmeteo_requests
import requests_cache
from retry_requests import retry
cache_session = requests_cache.CachedSession('.cache', expire_after=-1)
retry_session = retry(cache_session, retries=5, backoff_factor=0.2)
openmeteo = openmeteo_requests.Client(session=retry_session)
row = extreme.iloc[0]
params = {
            "latitude": row["begin_lat"],
            "longitude": row["begin_lon"],
            "start_date": row["begin_date_utc"],
            "end_date": row["end_date_utc"],
            "hourly": [
                "temperature_2m",
                "relative_humidity_2m",
                "dew_point_2m",
                "apparent_temperature",
                "precipitation",
                "rain",
                "snowfall",
                "snow_depth",
                "weather_code",
                "pressure_msl",
                "surface_pressure",
                "cloud_cover",
                "wind_speed_10m",
                "wind_direction_10m",
                "wind_gusts_10m",
                "soil_temperature_0_to_7cm",
                "soil_moisture_0_to_7cm"
            ]
        }


responses = openmeteo.weather_api(
    "https://archive-api.open-meteo.com/v1/archive",
    params=params
```

```
)
response = responses[0]

# Process hourly data
hourly = response.Hourly()
data = {
    "date": pd.date_range(
        start=pd.to_datetime(hourly.Time(), unit="s", utc=True),
        end=pd.to_datetime(hourly.TimeEnd(), unit="s", utc=True),
        freq=pd.Timedelta(seconds=hourly.Interval()),
        inclusive="left"
    ),
    "latitude": row["begin_lat"],
    "longitude": row["begin_lon"]
}

# Add all weather variables
for idx, var in enumerate(params["hourly"]):
    data[var] = hourly.Variables(idx).ValuesAsNumpy()
```

```
data["precipitation"]
```

```
row.index
```

```
import openmeteo_requests
import requests_cache
from retry_requests import retry
cache_session = requests_cache.CachedSession('.cache', expire_after=-1)
retry_session = retry(cache_session, retries=5, backoff_factor=0.2)
openmeteo = openmeteo_requests.Client(session=retry_session)
row = non_extreme.iloc[0]
params = {
            "latitude": row["begin_lat"],
            "longitude": row["begin_lon"],
            "start_date": row["begin_date_utc"],
            "end_date": row["end_date_utc"],
            "hourly": [
                "temperature_2m",
                "relative_humidity_2m",
                "dew_point_2m",
                "apparent_temperature",
                "precipitation",
                "rain",
                "snowfall",
                "snow_depth",
                "weather_code",
                "pressure_msl",
                "surface_pressure",
                "cloud_cover",
                "wind_speed_10m",
                "wind_direction_10m",
                "wind_gusts_10m",
                "soil_temperature_0_to_7cm",
```

```
                "soil_moisture_0_to_7cm"
            ]
        }


responses = openmeteo.weather_api(
    "https://archive-api.open-meteo.com/v1/archive",
    params=params
)
response = responses[0]

# Process hourly data
hourly = response.Hourly()
data = {
    "date": pd.date_range(
        start=pd.to_datetime(hourly.Time(), unit="s", utc=True),
        end=pd.to_datetime(hourly.TimeEnd(), unit="s", utc=True),
        freq=pd.Timedelta(seconds=hourly.Interval()),
        inclusive="left"
    ),
    "latitude": row["begin_lat"],
    "longitude": row["begin_lon"]
}

# Add all weather variables
for idx, var in enumerate(params["hourly"]):
    data[var] = hourly.Variables(idx).ValuesAsNumpy()
```

```
data["precipitation"]
```

```
import pandas as pd

merged_df = pd.read_csv(r"C:\Personal\Capstone\merged_df_copy.csv")
```

```
meteo_cols = [col for col in merged_df.columns if col.startswith("prev_72h")]
meteo_cols.remove("prev_72h_weather")
meteo_cols.remove("prev_72h_weather_code")
meteo_cols.remove("prev_72h_snow_depth")
meteo_cols.remove("prev_72h_snowfall")
```

```
import numpy as np
from tqdm import tqdm
import ast
for col in tqdm(meteo_cols):
    merged_df[col] = merged_df[col].apply(lambda x: np.array(ast.literal_eval(x)) if isinstanc
```

```
from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()
data_3d_X_train = np.stack(merged_df[meteo_cols].values.tolist(), axis=0)
# Reshape for scaling: (35000 * 27, 72)
```

```
reshaped_data_X_train = data_3d_X_train.reshape(-1, data_3d_X_train.shape[-1])
# Fit and transform
standardized_data_X_train = scaler.fit_transform(reshaped_data_X_train)


# Reshape back to (35000, 27, 72)
standardized_data_3d_train = standardized_data_X_train.reshape(data_3d_X_train.shape)
```

```
standardized_data_3d_train.shape
```

```
import tensorflow as tf
from tensorflow.keras import layers, models

def transformer_encoder(inputs, head_size, num_heads, ff_dim, dropout=0):
    # Normalization and Attention
    x = layers.LayerNormalization(epsilon=1e-6)(inputs)
    x = layers.MultiHeadAttention(
        key_dim=head_size, num_heads=num_heads, dropout=dropout
    )(x, x)
    x = layers.Dropout(dropout)(x)
    res = x + inputs

    # Feed Forward Part
    x = layers.LayerNormalization(epsilon=1e-6)(res)
    x = layers.Conv1D(filters=ff_dim, kernel_size=1, activation="relu")(x)
    x = layers.Dropout(dropout)(x)
    x = layers.Conv1D(filters=inputs.shape[-1], kernel_size=1)(x)
    return x + res

def build_model(
    input_shape,
    head_size,
    num_heads,
    ff_dim,
    num_transformer_blocks,
    mlp_units,
    dropout=0,
    mlp_dropout=0,
):
    inputs = layers.Input(shape=input_shape)
    x = inputs
    for _ in range(num_transformer_blocks):
        x = transformer_encoder(x, head_size, num_heads, ff_dim, dropout)

    x = layers.GlobalAveragePooling1D(data_format="channels_first")(x)
    for dim in mlp_units:
        x = layers.Dense(dim, activation="relu")(x)
        x = layers.Dropout(mlp_dropout)(x)
    outputs = layers.Dense(input_shape[0] * input_shape[1], activation="sigmoid")(x)
    outputs = layers.Reshape((input_shape[0], input_shape[1]))(outputs)
    return models.Model(inputs, outputs)


# Parameters
input_shape = (27, 72)
```

```python
head_size = 256
num_heads = 4
ff_dim = 4
num_transformer_blocks = 4
mlp_units = [128]
dropout = 0.25
mlp_dropout = 0.4

model = build_model(
    input_shape,
    head_size,
    num_heads,
    ff_dim,
    num_transformer_blocks,
    mlp_units,
    dropout=dropout,
    mlp_dropout=mlp_dropout,
)

model.compile(
    loss="mse",
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
)

# Assuming 'data' is your dataset with shape (35000, 27, 72)
history = model.fit(
    standardized_data_3d_train,
    standardized_data_3d_train,
    epochs=50,
    batch_size=64,
    validation_split=0.1,
)
```

```python
import numpy as np

# Get the reconstructed data
reconstructed_data = model.predict(standardized_data_3d_train)
# Calculate the Mean Squared Error (MSE) between the original and reconstructed data
mse = np.mean(np.square(standardized_data_3d_train - reconstructed_data), axis=(1, 2))
# Determine a threshold value for anomalies, e.g., mean + 3*std of the MSE
threshold = np.mean(mse) + 1 * np.std(mse)
# Identify anomalies
anomalies = mse > threshold
```

```python
t = 0
f = 0
for i in range(len(anomalies)):
    if anomalies[i] == True:
        t += 1
    else:
        f += 1
print(t, f)
```

```

```

show the data collection steps

show the data cleaning steps and data aligning steps

show the data labelling steps

show the EDA step

```
import pandas as pd

merged_df = pd.read_csv(r"C:\Personal\Capstone\merged_df_copy.csv")
```

```
import math
extreme = merged_df[merged_df['extreme'] == 1]
total = math.floor(1.5*extreme.shape[0])
non_extreme = merged_df[merged_df['extreme'] == 0].sample(total, random_state=1)

merged_df_new = pd.concat([extreme, non_extreme], axis=0)
```

```
merged_df_new["extreme"].shape
```

```
meteo_cols = [col for col in merged_df.columns if col.startswith("prev_72h")]
meteo_cols.remove("prev_72h_weather")
meteo_cols.remove("prev_72h_weather_code")
meteo_cols.remove("prev_72h_snow_depth")
meteo_cols.remove("prev_72h_snowfall")
```

```
meteo_cols
```

```
import numpy as np
from tqdm import tqdm
import ast
for col in tqdm(meteo_cols):
    merged_df[col] = merged_df[col].apply(lambda x: np.array(ast.literal_eval(x)) if isinstanc
```

```
features_column = pd.read_csv(r"C:\Personal\Capstone\features_column.csv")
```

```
features_column["features"].iloc[0]
```

```
merged_df
```

```
features_column
```

```
final_merged_df = pd.merge(merged_df, features_column, how="left", left_index=True, right_on='
```

```
final_merged_df.dropna(subset=["features"], inplace=True)
```

```
final_merged_df.shape
```

```
import concurrent.futures
import numpy as np
import torch
from transformers import ViTFeatureExtractor, ViTModel
from PIL import Image
import pandas as pd

# Assume final_merged_df, standardized_time_series_data, and image_paths are already defined.
# Example:
# final_merged_df = pd.DataFrame({'filenames': [['img1.jpg', 'img2.jpg'], ['img3.jpg'], ...],
# image_paths = [
#     "C:/Personal/Capstone/satellite_images/",
#     "C:\\Personal\\Capstone project\\Capstone project\\satellite_images\\satellite_images\\'
# ]

# Load feature extractor and model
model_name = 'DunnBC22/vit-base-patch16-224-in21k-weather-images-classification'
feature_extractor = ViTFeatureExtractor.from_pretrained(model_name)
model = ViTModel.from_pretrained(model_name)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
model.eval()
```

```
final_list = [f for f in ast.literal_eval(final_merged_df["filenames"].iloc[3546]) if "MODIS_T
```

```
final_list
```

```
features_column["features"].iloc[0]
```

```
import ast
def load_image(filename, image_paths):
    """
    Attempts to load an image from the provided paths.
    """

    for path in image_paths:
        try:
            full_path = path + filename
            image = Image.open(full_path).convert('RGB')
            return image
        except Exception as e:
            continue
    print(f"Error loading image {filename}")
    return None

def extract_features_for_row(image_list, image_paths, feature_extractor, model, device):
    """
    Given a list of image filenames for one row, extract features for each image,
    then concatenate the features into a single vector.
    """
```

```
    features_list = []
    for img in image_list:
        if img is None:
            feat = np.zeros(768)
        else:
            image = load_image(img, image_paths)
            if image is None:
                feat = np.zeros(768)
            else:
                inputs = feature_extractor(images=image, return_tensors="pt")
                inputs = {k: v.to(device) for k, v in inputs.items()}
                with torch.no_grad():
                    outputs = model(**inputs)
                # Extract the [CLS] token embedding which is of shape (1, 768)
                cls_embedding = outputs.last_hidden_state[:, 0, :]
                feat = cls_embedding.cpu().numpy().flatten()
        features_list.append(feat)
    # Concatenate features from all images (e.g., 3 images each of 768 dims become 2304 dims)
    return np.concatenate(features_list)

image_paths = [
    "C:/Personal/Capstone/satellite_images/",
    "C:\\Personal\\Capstone project\\Capstone project\\satellite_images\\satellite_images\\"
]
# Extract features in parallel and store them in the DataFrame.
first = extract_features_for_row(final_list, image_paths, feature_extractor, model, device)
```

```
first
```

```
print(final_merged_df["features"].iloc[3546])
print(first)
```

Try to Extract weather related features

```
import pandas as pd
import numpy as np
from joblib import Parallel, delayed
from tqdm.auto import tqdm
import multiprocessing as mp
import os

# Configuration
N_JOBS = min(os.cpu_count(), 8)  # Limit cores for memory safety
CHUNK_SIZE = 100  # Optimal for most datasets

# Feature calculation functions
def calculate_statistical(values):
    return {
        'mean': np.nanmean(values),
        'median': np.nanmedian(values),
        'min': np.nanmin(values),
        'max': np.nanmax(values),
        'std': np.nanstd(values),
        'variance': np.nanvar(values),
```

```python
        'skewness': pd.Series(values).skew(),
        'kurtosis': pd.Series(values).kurt()
    }

def calculate_temporal(values):
    features = {}
    if len(values) >= 24:
        slope, intercept = np.polyfit(range(len(values)), values, 1)
        features.update({
            'trend_slope': slope,
            'trend_intercept': intercept,
            'lag_24h': values[-24],
            'lag_48h': values[-48] if len(values) >= 48 else np.nan,
            'lag_72h': values[-72] if len(values) >= 72 else np.nan
        })
    else:
        features.update({k: np.nan for k in ['trend_slope', 'trend_intercept',
                                             'lag_24h', 'lag_48h', 'lag_72h']})
    return features

def calculate_rolling(values):
    features = {}
    if len(values) >= 24:
        s = pd.Series(values)
        window = s.rolling(24)
        features.update({
            'rolling_mean_24h': window.mean().iloc[-1],
            'rolling_std_24h': window.std().iloc[-1],
            'rolling_max_24h': window.max().iloc[-1],
            'rolling_min_24h': window.min().iloc[-1]
        })
    else:
        features.update({f'rolling_{stat}_24h': np.nan
                        for stat in ['mean', 'std', 'max', 'min']})
    return features

def calculate_domain(col_name, values):
    features = {}
    if 'temperature' in col_name:
        features['temp_diff'] = np.nanmax(values) - np.nanmin(values)
    elif 'precipitation' in col_name:
        features.update({
            'total_precip': np.nansum(values),
            'precip_hours': sum(x > 0 for x in values)
        })
    elif 'humidity' in col_name:
        features['humidity_range'] = np.nanmax(values) - np.nanmin(values)
    return features

# Main processing function
def process_row(row, columns):
    features = {}
    features['original_index'] = row.name
    for col in columns:
        values = row[col]
        # Statistical features
```

```
        stats = calculate_statistical(values)
        features.update({f'{col}_{k}': v for k, v in stats.items()})

        # Temporal features
        temporal = calculate_temporal(values)
        features.update({f'{col}_{k}': v for k, v in temporal.items()})

        # Rolling features
        rolling = calculate_rolling(values)
        features.update({f'{col}_{k}': v for k, v in rolling.items()})

        # Domain-specific features
        domain = calculate_domain(col, values)
        features.update({f'{col}_{k}': v for k, v in domain.items()})

    return features

# Parallel executor
def parallel_feature_extraction(df, columns):
    rows = [row for _, row in df.iterrows()]

    # Process in parallel with progress tracking
    results = Parallel(n_jobs=N_JOBS, batch_size=CHUNK_SIZE)(
        delayed(process_row)(row, columns)
        for row in tqdm(rows, desc='Processing rows', position=0)
    )

    return pd.DataFrame(results)

# Sample data generation and execution
if __name__ == '__main__':
    # Create test data (1000 rows × 3 columns)

    all_featires = merged_df[meteo_cols].copy()

    # Execute feature extraction
    final_df = parallel_feature_extraction(all_featires, meteo_cols)
    final_df = final_df.set_index('original_index')
    # Results verification
    print(f"\n✅ Processing complete!")
    print(f"Final DataFrame shape: {final_df.shape}")
    print("Sample features:")
    print(final_df.iloc[0, :5].to_string())  # Show first 5 features of first row
```

perform standardization and normalization

```
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Assuming final_df is your DataFrame of shape (35137, 468)
# It's a good idea to handle missing values; here I'm filling them with zeros as an example.
final_df_filled = final_df.fillna(0)

# Standardization: transform data to have mean 0 and standard deviation 1.
```

```
scaler_standard = StandardScaler()
standardized_data = scaler_standard.fit_transform(final_df_filled)
standardized_df = pd.DataFrame(standardized_data, index=final_df_filled.index, columns=final_c

# Normalization: scale data so that features are in the range [0, 1].
scaler_minmax = MinMaxScaler()
normalized_data = scaler_minmax.fit_transform(final_df_filled)
normalized_df = pd.DataFrame(normalized_data, index=final_df_filled.index, columns=final_df_fi

# Display first 5 rows of each result for verification.
print("Standardized Data (first 5 rows):")
print(standardized_df.head())

print("\nNormalized Data (first 5 rows):")
print(normalized_df.head())
```

perform PCA and other feature dimensionality reduction techniques

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import umap.umap_ as umap

# Assume normalized_df is your normalized DataFrame of shape (35137, 468)
# If needed, fill or drop any remaining missing values beforehand.
data = normalized_df.copy()

# ---------------------------
# PCA Dimensionality Reduction
# ---------------------------
# We'll reduce the data to 2 principal components.
pca = PCA(n_components=2)
pca_result = pca.fit_transform(data)
print("PCA explained variance ratio:", pca.explained_variance_ratio_)

# Create a DataFrame for PCA results.
pca_df = pd.DataFrame(data=pca_result, columns=['PC1', 'PC2'], index=data.index)

# ---------------------------
# UMAP Dimensionality Reduction
# ---------------------------
# UMAP parameters can be adjusted; here we reduce the data to 2 components.
umap_reducer = umap.UMAP(n_components=2, random_state=42)
umap_result = umap_reducer.fit_transform(data)

# Create a DataFrame for UMAP results.
umap_df = pd.DataFrame(data=umap_result, columns=['UMAP1', 'UMAP2'], index=data.index)

# ---------------------------
# Plotting the Results
# ---------------------------
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Scatter plot for PCA
axes[0].scatter(pca_df['PC1'], pca_df['PC2'], s=10, alpha=0.7)
```

```
axes[0].set_title("PCA Projection")
axes[0].set_xlabel("PC1")
axes[0].set_ylabel("PC2")

# Scatter plot for UMAP
axes[1].scatter(umap_df['UMAP1'], umap_df['UMAP2'], s=10, alpha=0.7, color='green')
axes[1].set_title("UMAP Projection")
axes[1].set_xlabel("UMAP1")
axes[1].set_ylabel("UMAP2")

plt.tight_layout()
plt.show()
```

try to do correlation test and see which featrres are corelated with each other to delete the redundunt ones

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
import seaborn as sns

# Assuming final_df is your DataFrame with shape (35137, 468)
# (Make sure it's numeric and handle missing values if needed)
final_df_filled = final_df.fillna(0)

# --------------------------------------
# Part 1: Visualizing Feature Clusters
# --------------------------------------
# Compute absolute correlation matrix
corr_matrix = final_df_filled.corr().abs()

# Compute the distance matrix as (1 - correlation)
# For hierarchical clustering, we need to convert our similarity measure into a distance measu
distance_matrix = 1 - corr_matrix

# Generate linkage matrix using average linkage method
linkage_matrix = linkage(distance_matrix, method='average')

# Plot dendrogram - note that labels might be overlapped due to a large number of features
plt.figure(figsize=(12, 8))
dendrogram(linkage_matrix, labels=corr_matrix.columns, leaf_rotation=90, leaf_font_size=8)
plt.title("Hierarchical Clustering Dendrogram of Features")
plt.xlabel("Feature")
plt.ylabel("Distance (1 - |Correlation|)")
plt.tight_layout()
plt.show()

# --------------------------------------------------
# Part 2: Automatic Feature Elimination by Correlation
# --------------------------------------------------
def drop_highly_correlated_features(df, correlation_threshold=0.95):
    """
    Drops features that are highly correlated. For each pair of features with a correlation
```

```
    above the threshold, one feature is removed.

    Args:
        df: DataFrame containing the features.
        correlation_threshold: Threshold above which features are considered redundant.

    Returns:
        A tuple (df_reduced, dropped_features) where:
         - df_reduced is the DataFrame after dropping features.
         - dropped_features is a list of feature names that were dropped.
    """
    corr_matrix = df.corr().abs()
    # Select the upper triangle of the correlation matrix
    upper_tri = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))

    # Find features with a correlation greater than the specified threshold
    drop_cols = [column for column in upper_tri.columns if any(upper_tri[column] > correlatior

    # Drop these features from the DataFrame
    df_reduced = df.drop(columns=drop_cols)
    return df_reduced, drop_cols

# Apply the function
reduced_df, dropped_features = drop_highly_correlated_features(final_df_filled, correlation_tł

print("Number of original features:", final_df_filled.shape[1])
print("Number of features after dropping highly correlated ones:", reduced_df.shape[1])
print("Dropped features:")
print(dropped_features)
```

Try traditional machine learning with this reduced df

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
from sklearn.model_selection import train_test_split, StratifiedKFold
# Enable experimental halving search in scikit-learn
from sklearn.experimental import enable_halving_search_cv  # noqa
from sklearn.model_selection import HalvingRandomSearchCV
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
from sklearn.svm import LinearSVC
# Import classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Optimized Gradient Boosting using LightGBM
import lightgbm as lgb

import warnings
warnings.filterwarnings('ignore')
```

```
# -------------------------
# 1. Data Preparation
# -------------------------
# Assume 'reduced_df' is your feature DataFrame and
# 'merged_df["extreme"]' is your target variable.
X = reduced_df.copy()
y = merged_df["extreme"].values

# Train-test split with stratification (90% training, 10% testing)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.10, random_state=42, stratify=y
)

print("Train and test split:")
print(f"Train set shape: {X_train.shape}")
print(f"Test set shape: {X_test.shape}")

# -------------------------
# 2. Define Cross-validation and Models
# -------------------------
# Use 3-fold CV to speed up tuning
cv_strategy = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

# Define models and their hyperparameter grids, including an optimized LightGBM-based Gradient
from sklearn.svm import LinearSVC

# Define models and their hyperparameter grids
models = {
    'Logistic Regression': {
        'model': LogisticRegression(class_weight='balanced', solver='liblinear', random_state=
        'params': {
            'C': [0.01, 0.1, 1, 10],
            'penalty': ['l2']
        }
    },
    'Decision Tree': {
        'model': DecisionTreeClassifier(class_weight='balanced', random_state=42),
        'params': {
            'max_depth': [None, 5, 10, 15],
            'min_samples_split': [2, 5, 10]
        }
    },
    'Random Forest': {
        'model': RandomForestClassifier(class_weight='balanced', n_estimators=200, random_stat
        'params': {
            'max_depth': [None, 5, 10, 15],
            'min_samples_split': [2, 5, 10]
        }
    },
    'Optimized Gradient Boosting': {
        'model': lgb.LGBMClassifier(class_weight='balanced', random_state=42),
        'params': {
            'num_leaves': [31, 50],
            'learning_rate': [0.05, 0.1],
            'n_estimators': [100, 200]
        }
```

```
    },
    'Support Vector Machine': {
        'model': LinearSVC(class_weight='balanced', random_state=42, max_iter=10000),
        'params': {
            'C': [0.1, 1, 10],
            'penalty': ['l2'],
            'loss': ['hinge']
        }
    }
}


results = {}

# -------------------------
# 3. Subsample Training Data for Fast Tuning
# -------------------------
# Use a subset (e.g., 30%) of the training data for hyperparameter tuning
# Define the sample fraction
sample_fraction = 0.3
X_train_reset = X_train.reset_index(drop=True)
# Sample indices
sample_indices = X_train_reset.sample(frac=sample_fraction, random_state=42).index

# Create the sampled datasets
X_train_sample = X_train_reset.loc[sample_indices]
y_train_sample = y_train[sample_indices]

print("\nStarting model training and hyperparameter tuning with HalvingRandomSearchCV:")

# -------------------------
# 4. Model Training with HalvingRandomSearchCV
# -------------------------
for model_name, config in tqdm(models.items(), desc="Models", total=len(models)):
    print(f"\n----- Training {model_name} -----")

    # Use HalvingRandomSearchCV for efficient hyperparameter tuning
    halving_search = HalvingRandomSearchCV(
        estimator=config['model'],
        param_distributions=config['params'],
        scoring='roc_auc',  # Using ROC-AUC for imbalanced data.
        cv=cv_strategy,
        n_jobs=-1,
        verbose=1,
        factor=2,           # Controls the aggressive down-selection
        max_resources='auto'
    )

    # Fit on the subsampled training data
    halving_search.fit(X_train_sample, y_train_sample)
    best_model = halving_search.best_estimator_

    print(f"Best parameters for {model_name}: {halving_search.best_params_}")
    print(f"Best cross validated ROC-AUC for {model_name}: {halving_search.best_score_:.4f}")

    # Evaluate on the full test set.
```

```python
    y_pred = best_model.predict(X_test)
    if hasattr(best_model, "predict_proba"):
        y_proba = best_model.predict_proba(X_test)[:, 1]
        auc = roc_auc_score(y_test, y_proba)
    else:
        auc = None

    print(f"\nEvaluation metrics for {model_name} on the test set:")
    print(classification_report(y_test, y_pred, digits=4))
    if auc is not None:
        print(f"Test ROC-AUC: {auc:.4f}")

    cm = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix:")
    print(cm)

    results[model_name] = {
        'best_model': best_model,
        'best_params': halving_search.best_params_,
        'best_cv_score': halving_search.best_score_,
        'classification_report': classification_report(y_test, y_pred, digits=4, output_dict=T
        'confusion_matrix': cm,
        'test_auc': auc
    }

print("\nAll models have been trained and evaluated with optimizations.")
```

Train models without dropping highly correlated features and train these models

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
from sklearn.model_selection import train_test_split, StratifiedKFold
# Enable experimental halving search in scikit-learn
from sklearn.experimental import enable_halving_search_cv  # noqa
from sklearn.model_selection import HalvingRandomSearchCV
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
from sklearn.svm import LinearSVC
# Import classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Optimized Gradient Boosting using LightGBM
import lightgbm as lgb

import warnings
warnings.filterwarnings('ignore')

# -------------------------
# 1. Data Preparation
```

```python
# -------------------------
# Assume 'reduced_df' is your feature DataFrame and
# 'merged_df["extreme"]' is your target variable.
X = final_df_filled.copy()
y = merged_df["extreme"].values

# Train-test split with stratification (90% training, 10% testing)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.10, random_state=42, stratify=y
)

print("Train and test split:")
print(f"Train set shape: {X_train.shape}")
print(f"Test set shape: {X_test.shape}")

# -------------------------
# 2. Define Cross-validation and Models
# -------------------------
# Use 3-fold CV to speed up tuning
cv_strategy = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

# Define models and their hyperparameter grids, including an optimized LightGBM-based Gradient
from sklearn.svm import LinearSVC

# Define models and their hyperparameter grids
models = {
    'Logistic Regression': {
        'model': LogisticRegression(class_weight='balanced', solver='liblinear', random_state=
        'params': {
            'C': [0.01, 0.1, 1, 10],
            'penalty': ['l2']
        }
    },
    'Decision Tree': {
        'model': DecisionTreeClassifier(class_weight='balanced', random_state=42),
        'params': {
            'max_depth': [None, 5, 10, 15],
            'min_samples_split': [2, 5, 10]
        }
    },
    'Random Forest': {
        'model': RandomForestClassifier(class_weight='balanced', n_estimators=200, random_stat
        'params': {
            'max_depth': [None, 5, 10, 15],
            'min_samples_split': [2, 5, 10]
        }
    },
    'Optimized Gradient Boosting': {
        'model': lgb.LGBMClassifier(class_weight='balanced', random_state=42),
        'params': {
            'num_leaves': [31, 50],
            'learning_rate': [0.05, 0.1],
            'n_estimators': [100, 200]
        }
    },
    'Support Vector Machine': {
```

```python
        'model': LinearSVC(class_weight='balanced', random_state=42, max_iter=10000),
        'params': {
            'C': [0.1, 1, 10],
            'penalty': ['l2'],
            'loss': ['hinge']
        }
    }
}


results = {}

# -------------------------
# 3. Subsample Training Data for Fast Tuning
# -------------------------
# Use a subset (e.g., 30%) of the training data for hyperparameter tuning
# Define the sample fraction
sample_fraction = 0.3
X_train_reset = X_train.reset_index(drop=True)
# Sample indices
sample_indices = X_train_reset.sample(frac=sample_fraction, random_state=42).index

# Create the sampled datasets
X_train_sample = X_train_reset.loc[sample_indices]
y_train_sample = y_train[sample_indices]

print("\nStarting model training and hyperparameter tuning with HalvingRandomSearchCV:")

# -------------------------
# 4. Model Training with HalvingRandomSearchCV
# -------------------------
for model_name, config in tqdm(models.items(), desc="Models", total=len(models)):
    print(f"\n----- Training {model_name} -----")

    # Use HalvingRandomSearchCV for efficient hyperparameter tuning
    halving_search = HalvingRandomSearchCV(
        estimator=config['model'],
        param_distributions=config['params'],
        scoring='roc_auc',  # Using ROC-AUC for imbalanced data.
        cv=cv_strategy,
        n_jobs=-1,
        verbose=1,
        factor=2,           # Controls the aggressive down-selection
        max_resources='auto'
    )

    # Fit on the subsampled training data
    halving_search.fit(X_train_sample, y_train_sample)
    best_model = halving_search.best_estimator_

    print(f"Best parameters for {model_name}: {halving_search.best_params_}")
    print(f"Best cross validated ROC-AUC for {model_name}: {halving_search.best_score_:.4f}")

    # Evaluate on the full test set.
    y_pred = best_model.predict(X_test)
    if hasattr(best_model, "predict_proba"):
```

```
        y_proba = best_model.predict_proba(X_test)[:, 1]
        auc = roc_auc_score(y_test, y_proba)
    else:
        auc = None

    print(f"\nEvaluation metrics for {model_name} on the test set:")
    print(classification_report(y_test, y_pred, digits=4))
    if auc is not None:
        print(f"Test ROC-AUC: {auc:.4f}")

    cm = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix:")
    print(cm)

    results[model_name] = {
        'best_model': best_model,
        'best_params': halving_search.best_params_,
        'best_cv_score': halving_search.best_score_,
        'classification_report': classification_report(y_test, y_pred, digits=4, output_dict=T
        'confusion_matrix': cm,
        'test_auc': auc
    }

print("\nAll models have been trained and evaluated with optimizations.")
```

Do PCA and umpa with gihher compoenets to see if it can capture more variance and train these classifcal models

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import umap.umap_ as umap

# Assume normalized_df is your normalized DataFrame of shape (35137, 468)
# If needed, fill or drop any remaining missing values beforehand.
data = normalized_df.copy()

# ---------------------------
# PCA Dimensionality Reduction
# ---------------------------
# We'll reduce the data to 2 principal components.
pca = PCA(n_components=35)
pca_result = pca.fit_transform(data)
print("PCA explained variance ratio:", pca.explained_variance_ratio_)

# Create a DataFrame for PCA results.
# pca_df = pd.DataFrame(data=pca_result, columns=['PC1', 'PC2'], index=data.index)
```

```
sum(pca.explained_variance_ratio_)  # Check how much variance is explained by the first 50 com
```

```
pca_result.copy()
```

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
from sklearn.model_selection import train_test_split, StratifiedKFold
# Enable experimental halving search in scikit-learn
from sklearn.experimental import enable_halving_search_cv  # noqa
from sklearn.model_selection import HalvingRandomSearchCV
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
from sklearn.svm import LinearSVC
# Import classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Optimized Gradient Boosting using LightGBM
import lightgbm as lgb

import warnings
warnings.filterwarnings('ignore')

# -------------------------
# 1. Data Preparation
# -------------------------
# Assume 'reduced_df' is your feature DataFrame and
# 'merged_df["extreme"]' is your target variable.
X = pca_result.copy()
y = merged_df["extreme"].values

# Train-test split with stratification (90% training, 10% testing)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.10, random_state=42, stratify=y
)

print("Train and test split:")
print(f"Train set shape: {X_train.shape}")
print(f"Test set shape: {X_test.shape}")

# -------------------------
# 2. Define Cross-validation and Models
# -------------------------
# Use 3-fold CV to speed up tuning
cv_strategy = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

# Define models and their hyperparameter grids, including an optimized LightGBM-based Gradient
from sklearn.svm import LinearSVC

# Define models and their hyperparameter grids
models = {
    'Logistic Regression': {
        'model': LogisticRegression(class_weight='balanced', solver='liblinear', random_state=
        'params': {
            'C': [0.01, 0.1, 1, 10],
            'penalty': ['l2']
```

```
        }
    },
    'Decision Tree': {
        'model': DecisionTreeClassifier(class_weight='balanced', random_state=42),
        'params': {
            'max_depth': [None, 5, 10, 15],
            'min_samples_split': [2, 5, 10]
        }
    },
    'Random Forest': {
        'model': RandomForestClassifier(class_weight='balanced', n_estimators=200, random_stat
        'params': {
            'max_depth': [None, 5, 10, 15],
            'min_samples_split': [2, 5, 10]
        }
    },
    'Optimized Gradient Boosting': {
        'model': lgb.LGBMClassifier(class_weight='balanced', random_state=42),
        'params': {
            'num_leaves': [31, 50],
            'learning_rate': [0.05, 0.1],
            'n_estimators': [100, 200]
        }
    },
    'Support Vector Machine': {
        'model': LinearSVC(class_weight='balanced', random_state=42, max_iter=10000),
        'params': {
            'C': [0.1, 1, 10],
            'penalty': ['l2'],
            'loss': ['hinge']
        }
    }
}


results = {}


print("\nStarting model training and hyperparameter tuning with HalvingRandomSearchCV:")

# -------------------------
# 4. Model Training with HalvingRandomSearchCV
# -------------------------
for model_name, config in tqdm(models.items(), desc="Models", total=len(models)):
    print(f"\n----- Training {model_name} -----")

    # Use HalvingRandomSearchCV for efficient hyperparameter tuning
    halving_search = HalvingRandomSearchCV(
        estimator=config['model'],
        param_distributions=config['params'],
        scoring='roc_auc',  # Using ROC-AUC for imbalanced data.
        cv=cv_strategy,
        n_jobs=-1,
        verbose=1,
        factor=2,          # Controls the aggressive down-selection
        max_resources='auto'
```

```
    )

    # Fit on the subsampled training data
    halving_search.fit(X_train, y_train)
    best_model = halving_search.best_estimator_

    print(f"Best parameters for {model_name}: {halving_search.best_params_}")
    print(f"Best cross validated ROC-AUC for {model_name}: {halving_search.best_score_:.4f}")

    # Evaluate on the full test set.
    y_pred = best_model.predict(X_test)
    if hasattr(best_model, "predict_proba"):
        y_proba = best_model.predict_proba(X_test)[:, 1]
        auc = roc_auc_score(y_test, y_proba)
    else:
        auc = None

    print(f"\nEvaluation metrics for {model_name} on the test set:")
    print(classification_report(y_test, y_pred, digits=4))
    if auc is not None:
        print(f"Test ROC-AUC: {auc:.4f}")

    cm = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix:")
    print(cm)

    results[model_name] = {
        'best_model': best_model,
        'best_params': halving_search.best_params_,
        'best_cv_score': halving_search.best_score_,
        'classification_report': classification_report(y_test, y_pred, digits=4, output_dict=T
        'confusion_matrix': cm,
        'test_auc': auc
    }

print("\nAll models have been trained and evaluated with optimizations.")
```

Do features selection using random forest, selectkbest and l1, l2 regiraliztion to select important features

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

X = final_df_filled.copy()
y = merged_df["extreme"].values

# Train-test split with stratification (90% training, 10% testing)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.10, random_state=42, stratify=y
)

rf = RandomForestClassifier(n_estimators=100, random_state=42)
```

```python
rf.fit(X_train, y_train)

selector = SelectFromModel(rf, threshold="mean", max_features=35)
X_train_selected = selector.fit_transform(X_train, y_train)
X_test_selected = selector.transform(X_test)
```

```python
from sklearn.feature_selection import SelectKBest, f_classif

# Initialize SelectKBest with ANOVA F-value
selector = SelectKBest(score_func=f_classif, k=35)
X_train_selected = selector.fit_transform(X_train, y_train)
X_test_selected = selector.transform(X_test)

# Get the selected feature indices
selected_indices = selector.get_support(indices=True)
selected_features = data.columns[selected_indices]
print(f"Selected features: {selected_features}")
```

```python
def tain_models(df):

    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
    from tqdm.auto import tqdm
    from sklearn.model_selection import train_test_split, StratifiedKFold
    # Enable experimental halving search in scikit-learn
    from sklearn.experimental import enable_halving_search_cv  # noqa
    from sklearn.model_selection import HalvingRandomSearchCV
    from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
    from sklearn.svm import LinearSVC
    # Import classifiers
    from sklearn.linear_model import LogisticRegression
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.svm import SVC

    # Optimized Gradient Boosting using LightGBM
    import lightgbm as lgb

    import warnings
    warnings.filterwarnings('ignore')

    # -------------------------
    # 1. Data Preparation
    # -------------------------
    # Assume 'reduced_df' is your feature DataFrame and
    # 'merged_df["extreme"]' is your target variable.
    X = df.copy()
    y = merged_df["extreme"].values

    # Train-test split with stratification (90% training, 10% testing)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.10, random_state=42, stratify=y
    )
```

```
print("Train and test split:")
print(f"Train set shape: {X_train.shape}")
print(f"Test set shape: {X_test.shape}")

# -------------------------
# 2. Define Cross-validation and Models
# -------------------------
# Use 3-fold CV to speed up tuning
cv_strategy = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

# Define models and their hyperparameter grids, including an optimized LightGBM-based Grad
from sklearn.svm import LinearSVC

# Define models and their hyperparameter grids
models = {
    'Logistic Regression': {
        'model': LogisticRegression(class_weight='balanced', solver='liblinear', random_st
        'params': {
            'C': [0.01, 0.1, 1, 10],
            'penalty': ['l2']
        }
    },
    'Decision Tree': {
        'model': DecisionTreeClassifier(class_weight='balanced', random_state=42),
        'params': {
            'max_depth': [None, 5, 10, 15],
            'min_samples_split': [2, 5, 10]
        }
    },
    'Random Forest': {
        'model': RandomForestClassifier(class_weight='balanced', n_estimators=200, random_
        'params': {
            'max_depth': [None, 5, 10, 15],
            'min_samples_split': [2, 5, 10]
        }
    },
    'Optimized Gradient Boosting': {
        'model': lgb.LGBMClassifier(class_weight='balanced', random_state=42),
        'params': {
            'num_leaves': [31, 50],
            'learning_rate': [0.05, 0.1],
            'n_estimators': [100, 200]
        }
    },
    'Support Vector Machine': {
        'model': LinearSVC(class_weight='balanced', random_state=42, max_iter=10000),
        'params': {
            'C': [0.1, 1, 10],
            'penalty': ['l2'],
            'loss': ['hinge']
        }
    }
}


results = {}
```

```python
print("\nStarting model training and hyperparameter tuning with HalvingRandomSearchCV:")

# -------------------------
# 4. Model Training with HalvingRandomSearchCV
# -------------------------
for model_name, config in tqdm(models.items(), desc="Models", total=len(models)):
    print(f"\n----- Training {model_name} -----")

    # Use HalvingRandomSearchCV for efficient hyperparameter tuning
    halving_search = HalvingRandomSearchCV(
        estimator=config['model'],
        param_distributions=config['params'],
        scoring='roc_auc',  # Using ROC-AUC for imbalanced data.
        cv=cv_strategy,
        n_jobs=-1,
        verbose=1,
        factor=2,           # Controls the aggressive down-selection
        max_resources='auto'
    )

    # Fit on the subsampled training data
    halving_search.fit(X_train, y_train)
    best_model = halving_search.best_estimator_

    print(f"Best parameters for {model_name}: {halving_search.best_params_}")
    print(f"Best cross validated ROC-AUC for {model_name}: {halving_search.best_score_:.4f

    # Evaluate on the full test set.
    y_pred = best_model.predict(X_test)
    if hasattr(best_model, "predict_proba"):
        y_proba = best_model.predict_proba(X_test)[:, 1]
        auc = roc_auc_score(y_test, y_proba)
    else:
        auc = None

    print(f"\nEvaluation metrics for {model_name} on the test set:")
    print(classification_report(y_test, y_pred, digits=4))
    if auc is not None:
        print(f"Test ROC-AUC: {auc:.4f}")

    cm = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix:")
    print(cm)

    results[model_name] = {
        'best_model': best_model,
        'best_params': halving_search.best_params_,
        'best_cv_score': halving_search.best_score_,
        'classification_report': classification_report(y_test, y_pred, digits=4, output_di
        'confusion_matrix': cm,
        'test_auc': auc
    }

print("\nAll models have been trained and evaluated with optimizations.")
```

```
    return results
selectd_features_result_1 = tain_models(final_df_filled[selected_features])
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SelectFromModel

# Initialize Logistic Regression with L1 penalty
model = LogisticRegression(penalty='l1', solver='liblinear', random_state=42)
model.fit(X_train, y_train)

# Select features based on L1 regularization
selector = SelectFromModel(model, threshold="mean", max_features=35)
X_train_selected = selector.transform(X_train)
X_test_selected = selector.transform(X_test)

# Get the selected feature indices
selected_indices = selector.get_support(indices=True)
selected_features = data.columns[selected_indices]
print(f"Selected features: {selected_features}")
```

```
selected_features_result_2 = tain_models(final_df_filled[selected_features])
```

```
from sklearn.linear_model import Ridge
from sklearn.feature_selection import SelectFromModel

# Initialize Ridge Regression with L2 penalty
model = Ridge(alpha=1.0)
model.fit(X_train, y_train)

# Select features based on feature importance
selector = SelectFromModel(model, threshold="mean", max_features=35)
X_train_selected = selector.transform(X_train)
X_test_selected = selector.transform(X_test)

# Get the selected feature indices
selected_indices = selector.get_support(indices=True)
selected_features = data.columns[selected_indices]
print(f"Selected features: {selected_features}")
```

```
selected_features_result_3 = tain_models(final_df_filled[selected_features])
```

Do feature selectoin on standadized data

```
standardized_df.shape
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

X = standardized_df.copy()
y = merged_df["extreme"].values

# Train-test split with stratification (90% training, 10% testing)
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.10, random_state=42, stratify=y
)

rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

selector = SelectFromModel(rf, threshold="mean", max_features=35)
X_train_selected = selector.fit_transform(X_train, y_train)
X_test_selected = selector.transform(X_test)
```

```
X_test_selected.shape
```

```
from sklearn.feature_selection import SelectKBest, f_classif

# Initialize SelectKBest with ANOVA F-value
selector = SelectKBest(score_func=f_classif, k=35)
X_train_selected = selector.fit_transform(X_train, y_train)
X_test_selected = selector.transform(X_test)

# Get the selected feature indices
selected_indices = selector.get_support(indices=True)
selected_features = data.columns[selected_indices]
print(f"Selected features: {selected_features}")
```

```
selected_features_standadrized_results = tain_models(standardized_df[selected_features])
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SelectFromModel

# Initialize Logistic Regression with L1 penalty
model = LogisticRegression(penalty='l1', solver='liblinear', random_state=42)
model.fit(X_train, y_train)

# Select features based on L1 regularization
selector = SelectFromModel(model, threshold="mean", max_features=35)
X_train_selected = selector.transform(X_train)
X_test_selected = selector.transform(X_test)

# Get the selected feature indices
selected_indices = selector.get_support(indices=True)
selected_features = data.columns[selected_indices]
print(f"Selected features: {selected_features}")
```

```
from sklearn.linear_model import Ridge
from sklearn.feature_selection import SelectFromModel

# Initialize Ridge Regression with L2 penalty
model = Ridge(alpha=1.0)
model.fit(X_train, y_train)

# Select features based on feature importance
selector = SelectFromModel(model, threshold="mean", max_features=35)
X_train_selected = selector.transform(X_train)
X_test_selected = selector.transform(X_test)
```

```
# Get the selected feature indices
selected_indices = selector.get_support(indices=True)
selected_features = data.columns[selected_indices]
print(f"Selected features: {selected_features}")
```

```
selected_features_standadrized_results_e = tain_models(standardized_df[selected_features])
```

Try wihtout engineered features, just scaled raw data

```
# Train-test split with stratification (90% training, 10% testing)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    merged_df[meteo_cols], merged_df["extreme"].values, test_size=0.10, random_state=42, strat
)
```

```
data_3d_X_train = np.stack(X_train.values.tolist(), axis=0)
data_3d_X_test = np.stack(X_test.values.tolist(), axis=0)
```

```
data_3d_X_train.shape
```

```
data_3d_X_train = data_3d_X_train[:,:,-12:]
```

```
data_3d_X_test = data_3d_X_test[:,:,-12:]
```

```
data_3d_X_train.shape
```

```
from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Reshape for scaling: (35000 * 27, 72)
reshaped_data_X_train = data_3d_X_train.reshape(-1, data_3d_X_train.shape[-1])
reshaped_data_X_test = data_3d_X_test.reshape(-1, data_3d_X_test.shape[-1])
# Fit and transform
standardized_data_X_train = scaler.fit_transform(reshaped_data_X_train)
standardized_data_X_test = scaler.transform(reshaped_data_X_test)

# Reshape back to (35000, 27, 72)
standardized_data_3d_train = standardized_data_X_train.reshape(data_3d_X_train.shape)
standardized_data_3d_test = standardized_data_X_test.reshape(data_3d_X_test.shape)
```

```
standardized_data_3d_train.shape
```

```
def train_models_original(X_train, y_train, X_test, y_test):

    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
    from tqdm.auto import tqdm
    from sklearn.model_selection import train_test_split, StratifiedKFold
```

```
# Enable experimental halving search in scikit-learn
from sklearn.experimental import enable_halving_search_cv  # noqa
from sklearn.model_selection import HalvingRandomSearchCV
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
from sklearn.svm import LinearSVC
# Import classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Optimized Gradient Boosting using LightGBM
import lightgbm as lgb

import warnings
warnings.filterwarnings('ignore')

# -------------------------
# 1. Data Preparation
# -------------------------
# Assume 'reduced_df' is your feature DataFrame and
# 'merged_df["extreme"]' is your target variable.



print("Train and test split:")
print(f"Train set shape: {X_train.shape}")
print(f"Test set shape: {X_test.shape}")

# -------------------------
# 2. Define Cross-validation and Models
# -------------------------
# Use 3-fold CV to speed up tuning
cv_strategy = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

# Define models and their hyperparameter grids, including an optimized LightGBM-based Grad
from sklearn.svm import LinearSVC

# Define models and their hyperparameter grids
models = {
    'Logistic Regression': {
        'model': LogisticRegression(class_weight='balanced', solver='liblinear', random_st
        'params': {
            'C': [0.01, 0.1, 1, 10],
            'penalty': ['l2']
        }
    },
    'Decision Tree': {
        'model': DecisionTreeClassifier(class_weight='balanced', random_state=42),
        'params': {
            'max_depth': [None, 5, 10, 15],
            'min_samples_split': [2, 5, 10]
        }
    },
    'Random Forest': {
        'model': RandomForestClassifier(class_weight='balanced', n_estimators=200, random_
```

```
        'params': {
            'max_depth': [None, 5, 10, 15],
            'min_samples_split': [2, 5, 10]
        }
    },
    'Optimized Gradient Boosting': {
        'model': lgb.LGBMClassifier(class_weight='balanced', random_state=42),
        'params': {
            'num_leaves': [31, 50],
            'learning_rate': [0.05, 0.1],
            'n_estimators': [100, 200]
        }
    },
    'Support Vector Machine': {
        'model': LinearSVC(class_weight='balanced', random_state=42, max_iter=10000),
        'params': {
            'C': [0.1, 1, 10],
            'penalty': ['l2'],
            'loss': ['hinge']
        }
    }
}


results = {}


print("\nStarting model training and hyperparameter tuning with HalvingRandomSearchCV:")

# ------------------------
# 4. Model Training with HalvingRandomSearchCV
# ------------------------
for model_name, config in tqdm(models.items(), desc="Models", total=len(models)):
    print(f"\n----- Training {model_name} -----")

    # Use HalvingRandomSearchCV for efficient hyperparameter tuning
    halving_search = HalvingRandomSearchCV(
        estimator=config['model'],
        param_distributions=config['params'],
        scoring='roc_auc',  # Using ROC-AUC for imbalanced data.
        cv=cv_strategy,
        n_jobs=-1,
        verbose=1,
        factor=2,           # Controls the aggressive down-selection
        max_resources='auto'
    )

    # Fit on the subsampled training data
    halving_search.fit(X_train, y_train)
    best_model = halving_search.best_estimator_

    print(f"Best parameters for {model_name}: {halving_search.best_params_}")
    print(f"Best cross validated ROC-AUC for {model_name}: {halving_search.best_score_:.4f

    # Evaluate on the full test set.
    y_pred = best_model.predict(X_test)
```

```
        if hasattr(best_model, "predict_proba"):
            y_proba = best_model.predict_proba(X_test)[:, 1]
            auc = roc_auc_score(y_test, y_proba)
        else:
            auc = None

        print(f"\nEvaluation metrics for {model_name} on the test set:")
        print(classification_report(y_test, y_pred, digits=4))
        if auc is not None:
            print(f"Test ROC-AUC: {auc:.4f}")

        cm = confusion_matrix(y_test, y_pred)
        print("Confusion Matrix:")
        print(cm)

        results[model_name] = {
            'best_model': best_model,
            'best_params': halving_search.best_params_,
            'best_cv_score': halving_search.best_score_,
            'classification_report': classification_report(y_test, y_pred, digits=4, output_di
            'confusion_matrix': cm,
            'test_auc': auc
        }

    print("\nAll models have been trained and evaluated with optimizations.")
    return results
```

```
standardized_data_3d_train_flattened = standardized_data_3d_train.reshape(standardized_data_3c
standardized_data_3d_test_flattened = standardized_data_3d_test.reshape(standardized_data_3d_t


import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import umap.umap_ as umap

# Assume normalized_df is your normalized DataFrame of shape (35137, 468)
# If needed, fill or drop any remaining missing values beforehand.

# ---------------------------
# PCA Dimensionality Reduction
# ---------------------------
# We'll reduce the data to 2 principal components.
pca = PCA(n_components=100)
pca_result_train = pca.fit_transform(standardized_data_3d_train_flattened)
pca_result_test = pca.transform(standardized_data_3d_test_flattened)
print("PCA explained variance ratio:", pca.explained_variance_ratio_)
print("Total variance explained: ", sum(pca.explained_variance_ratio_))
```

```
results_original = train_models_original(pca_result_train, y_train, pca_result_test, y_test)
```

Do undersampling, oversampling, smote, Adaysn and train these models

```
%pip install smote-variants==1.0.1
```

```
.shape
```

```
import smote_variants
print(smote_variants.__version__)
```

```
from smote_variants import SMOTE_IPF, G_SMOTE, SMOTE_ENN

# Reshape to 2D (samples, features*timesteps)
X_2d = standardized_data_3d_train.reshape(-1, 27*72)

# 1. SMOTE_IPF (Imitates physical weather patterns)
# oversampler_ipf = SMOTE_IPF(
#     n_components=35,
#     n_neighbors=12,  # Matches 72h temporal window (12h × 6)
#     proportion=1.0,  # 100% of the minority class

# )
# X_res_ipf, y_res_ipf = oversampler_ipf.sample(X_2d, y_train)

# # 2. G-SMOTE (Geometric SMOTE for multivariate series)
# oversampler_g = G_SMOTE(
#     n_neighbors=6,    # 6-hour temporal blocks
#     deformation_factor=0.5
# )
# X_res_g, y_res_g = oversampler_ipf.sample(X_2d, y_train)
import time
from tqdm import tqdm

class ProgressSMOTE(SMOTE_IPF):
    def _sample(self, X, y):
        with tqdm(total=len(X)) as pbar:
            for _ in super()._sample(X, y):
                pbar.update(1)
                yield

oversampler = ProgressSMOTE()
X_res, y_res = oversampler.sample(X_2d, y_train)
```

```
standardized_data_3d_train.shape
```

```
import numpy as np
import pandas as pd
from sklearn.neighbors import NearestNeighbors
import random
import math
from tqdm import tqdm # Import tqdm

# Assuming tSmote.py is in the same directory or accessible in your Python path
# We won't directly use most functions, but will adapt the SMOTE logic.
# import tSmote as ts # We'll extract the core logic instead

# --- Configuration ---
```

```
RANDOM_SEED = 42
N_NEIGHBORS = 5 # K value for SMOTE, ensure N_NEIGHBORS < number of minority samples

random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

# --- Load Your Data ---
# This is a placeholder. Replace with your actual data loading.
# Example: df = pd.read_csv('your_weather_data.csv')
#          weather_data = ... # extract the (35000, 27, 72) numpy array
#          labels = df['extreme'].values # extract the (35000,) labels array

# For demonstration, let's create dummy data with the specified shape
print("Creating dummy data...")
# Use a smaller sample size for quick testing, increase for your real data
n_samples = 31623 # Use 35000 for your actual data
n_features = 27
n_timesteps = 72

# Simulate imbalanced data (e.g., 10% minority class)
minority_fraction = 0.1
n_minority = int(n_samples * minority_fraction)
n_majority = n_samples - n_minority

weather_data = standardized_data_3d_train
labels = y_train
np.random.shuffle(labels) # Shuffle labels to mix classes

print(f"Dummy data shape: {weather_data.shape}")
print(f"Labels shape: {labels.shape}")
print(f"Minority samples: {n_minority}, Majority samples: {n_majority}")
print("-" * 30)

# --- Separate Data by Class ---
minority_data = weather_data[labels == 1]
majority_data = weather_data[labels == 0]

n_min_samples = minority_data.shape[0]

if n_min_samples <= N_NEIGHBORS:
    raise ValueError(f"Number of minority samples ({n_min_samples}) must be greater than N_NEI

# --- Calculate Number of Synthetic Samples Needed ---
n_synthetic_needed = n_majority - n_min_samples
if n_synthetic_needed <= 0:
    print("Minority class is already balanced or larger. No oversampling needed.")
    synthetic_data_final = np.zeros((0, n_features, n_timesteps)) # Create empty array if no s
else:
    print(f"Need to generate {n_synthetic_needed} synthetic minority samples.")

print("-" * 30)

# --- SMOTE Logic (Adapted, Vector-wise per time step) ---

def smote_time_step(data_at_t, n_to_generate, k_neighbors):
    """
```

```
    Applies SMOTE to the data slice corresponding to a single time step.

    Args:
        data_at_t (np.ndarray): Data for minority class at a specific time step (shape: n_minc
        n_to_generate (int): Number of synthetic samples to generate for this time step.
        k_neighbors (int): Number of nearest neighbors for SMOTE.

    Returns:
        np.ndarray: Array of synthetic samples for this time step (shape: n_to_generate, n_fea
    """
    n_minority_samples, n_features = data_at_t.shape
    synthetic_samples = np.zeros((n_to_generate, n_features))

    if n_minority_samples <= k_neighbors:
        # Handle edge case: not enough samples for specified k
        # Option 1: Reduce k
        k_neighbors = n_minority_samples - 1
        if k_neighbors <= 0:
            # Just duplicate the single sample if k becomes 0 or less
            indices = np.random.choice(n_minority_samples, n_to_generate, replace=True)
            return data_at_t[indices]

    nn = NearestNeighbors(n_neighbors=k_neighbors + 1).fit(data_at_t) # Fit with k+1 to incluc

    for i in range(n_to_generate):
        # 1. Choose a random minority sample
        idx = random.randint(0, n_minority_samples - 1)
        sample = data_at_t[idx]

        # 2. Find its k nearest neighbors
        distances, indices = nn.kneighbors(sample.reshape(1, -1))
        # indices[0][0] is the sample itself, neighbors are indices[0][1:]
        neighbor_indices = indices[0][1:k_neighbors+1] # Select exactly k neighbors

        if not len(neighbor_indices): # Should only happen if k became 0
            synthetic_samples[i] = sample # Just duplicate if no neighbours left
            continue

        # 3. Choose one neighbor randomly
        chosen_neighbor_idx = random.choice(neighbor_indices)
        neighbor = data_at_t[chosen_neighbor_idx]

        # 4. Generate synthetic sample
        gap = random.random()
        synthetic_samples[i] = sample + gap * (neighbor - sample)

    return synthetic_samples

# --- Generate Synthetic Data Time Step by Time Step ---
if n_synthetic_needed > 0: # Only run if we need synthetic samples
    print("Generating synthetic data using adapted tSMOTE (vector-wise per time step)...")
    all_synthetic_data_t = [] # List to hold synthetic data for each timestep

    # Wrap range(n_timesteps) with tqdm for progress bar
    for t in tqdm(range(n_timesteps), desc="Generating Synthetic Data (Time Steps)"):
        data_slice_at_t = minority_data[:, :, t] # Shape: (n_min_samples, n_features)
```

```
        # Generate n_synthetic_needed at *each* step to create n_synthetic_needed new series
        if n_min_samples > 1 : # Need at least 2 samples for SMOTE
           synthetic_slice = smote_time_step(data_slice_at_t, n_synthetic_needed, N_NEIGHBORS)
        else: # Handle case with 0 or 1 minority sample at this step
           if n_min_samples == 1:
               synthetic_slice = np.tile(data_slice_at_t, (n_synthetic_needed, 1))
           else:
               synthetic_slice = np.zeros((n_synthetic_needed, n_features)) # Or handle differ
               # Add a warning if desired, but might be too noisy in a loop
               # print(f"Warning: 0 minority samples at timestep {t}. Generating zeros.")

        all_synthetic_data_t.append(synthetic_slice) # List of arrays [ (N_synth, feats), (N_s

    print("Synthetic data generation complete.")
    print("-" * 30)

    # --- Reassemble Synthetic Data ---
    print("Reassembling synthetic data...")
    synthetic_data_final = np.zeros((n_synthetic_needed, n_features, n_timesteps))

    for t in range(n_timesteps):
        synthetic_data_final[:, :, t] = all_synthetic_data_t[t]

    print(f"Shape of final synthetic data: {synthetic_data_final.shape}")
    print("-" * 30)

else: # Case where no synthetic samples were needed
    synthetic_data_final = np.zeros((0, n_features, n_timesteps))


# --- Combine Original Minority and Synthetic Data ---
oversampled_minority_data = np.concatenate((minority_data, synthetic_data_final), axis=0)
oversampled_minority_labels = np.ones(oversampled_minority_data.shape[0], dtype=int)

print(f"Shape of oversampled minority data: {oversampled_minority_data.shape}")

# --- (Optional) Combine with Majority Data ---
final_weather_data = np.concatenate((majority_data, oversampled_minority_data), axis=0)
final_labels = np.concatenate((labels[labels == 0], oversampled_minority_labels), axis=0)

# Optionally shuffle the final combined dataset
shuffle_indices = np.random.permutation(len(final_labels))
final_weather_data = final_weather_data[shuffle_indices]
final_labels = final_labels[shuffle_indices]


print(f"Shape of final balanced data: {final_weather_data.shape}")
print(f"Shape of final balanced labels: {final_labels.shape}")
print(f"Final class distribution: {np.bincount(final_labels)}")

# --- Now you can use 'final_weather_data' and 'final_labels' for your model training ---
```

```
## most efficiently save the variables
standardized_data_3d_train.shape
```

```
sum(y_train)/len(y_train), sum(y_test)/len(y_test)
```

```
import numpy as np
import pandas as pd
import importlib
# from sklearn.neighbors import NearestNeighbors # No longer needed directly here
import random
# import math # Not directly needed here
from tqdm import tqdm # For progress bar

# --- IMPORTANT: Make sure tSmote.py is in the same directory ---
# --- or that its path is added to your Python environment ---
try:
    import tSmote as ts
    print("Successfully imported tSmote.py")
except ImportError:
    print("ERROR: Could not import tSmote.py. Make sure it's in the correct directory or Pytho
    exit()
importlib.reload(ts)
# --- Configuration ---
RANDOM_SEED = 42
N_NEIGHBORS = 5 # K value for SMOTE, ensure N_NEIGHBORS < number of minority samples

# Seed setting - tSmote.py already sets np.random.seed(0) internally
# We might want to control the standard random module seed as well.
random.seed(RANDOM_SEED)
# Note: ts.generateTimePoints uses np.random internally, which is seeded to 0 in tSmote.py
# If you need different seeding for numpy within ts.generateTimePoints, you'd modify tSmote.py
# or re-seed numpy *just before* calling it, knowing it might affect reproducibility if
# other parts of your code rely on numpy's random state. For simplicity, we'll rely
# on the internal seed(0) for now, but be aware of this.
# np.random.seed(RANDOM_SEED) # Overridden by tSmote.py if imported after

# --- Load Your Data ---
# Placeholder - Replace with your actual data loading
print("Creating dummy data...")
n_samples = 7782 # Use 35000 for your actual data
n_features = 27
n_timesteps = 72

minority_fraction = sum(y_train)/len(y_train)
n_minority = int(n_samples * minority_fraction)
n_majority = n_samples - n_minority

weather_data = standardized_data_3d_train.shape[0]
labels = y_train
np.random.shuffle(labels)

print(f"Dummy data shape: {weather_data.shape}")
print(f"Labels shape: {labels.shape}")
print(f"Minority samples: {n_minority}, Majority samples: {n_majority}")
print("-" * 30)

# --- Separate Data by Class ---
minority_data = weather_data[labels == 1]
```

```
majority_data = weather_data[labels == 0]

n_min_samples = minority_data.shape[0]

if n_min_samples <= N_NEIGHBORS:
    # generateTimePoints might handle this by reducing k, but let's check upfront
    print(f"Warning: Number of minority samples ({n_min_samples}) is less than or equal to N_N
    if n_min_samples <= 1:
         raise ValueError(f"Cannot perform SMOTE with {n_min_samples} minority sample(s). Need


# --- Calculate Number of Synthetic Samples Needed ---
n_synthetic_needed = n_majority - n_min_samples
if n_synthetic_needed <= 0:
    print("Minority class is already balanced or larger. No oversampling needed.")
    synthetic_data_final = np.zeros((0, n_features, n_timesteps)) # Empty array
else:
    print(f"Need to generate {n_synthetic_needed} synthetic minority samples.")

print("-" * 30)


# --- Prepare Input Data Structure for generateTimePoints ---
# We need a list of lists, where each inner list contains the feature vectors
# for all minority samples at a specific time step.
minority_time_slices = []
print("Structuring minority data into time slices...")
for t in tqdm(range(n_timesteps), desc="Preparing Time Slices"):
    # Extract data for time step t: shape (n_minority_samples, n_features)
    data_slice_at_t = minority_data[:, :, t]
    # Convert to list of lists (as expected by generateTimePoints)
    minority_time_slices.append(data_slice_at_t.tolist())

print(f"Created {len(minority_time_slices)} time slices for minority data.")
print("-" * 30)


# --- Generate Synthetic Data using tSmote.generateTimePoints ---
if n_synthetic_needed > 0:
    print(f"Generating {n_synthetic_needed} synthetic samples per time step using ts.generateT
    # nPoints parameter is the number of synthetic samples PER time slice.
    # Since we want n_synthetic_needed *new series*, we generate this many per slice.
    # Note: generateTimePoints doesn't have a built-in progress bar for its internal loops.
    tSliceSyn = ts.generateTimePoints(minority_time_slices, nPoints=n_synthetic_needed, nNeigh
    print("Synthetic data generation complete (using tSmote.py function).")
    print("-" * 30)

    # --- Reassemble Synthetic Data ---
    # Output tSliceSyn is List[List[List[float]]], specifically List_timeslice[List_synthetic_
    # Shape of tSliceSyn[t] is (n_synthetic_needed, n_features)
    print("Reassembling synthetic data...")
    synthetic_data_final = np.zeros((n_synthetic_needed, n_features, n_timesteps))

    for t in tqdm(range(n_timesteps), desc="Reassembling Data"):
        # Convert the inner list of lists to a numpy array and assign
        synthetic_data_final[:, :, t] = np.array(tSliceSyn[t])
```

```python
    print(f"Shape of final synthetic data: {synthetic_data_final.shape}")
    print("-" * 30)

else: # Case where no synthetic samples were needed
    synthetic_data_final = np.zeros((0, n_features, n_timesteps))

# --- Combine Original Minority and Synthetic Data ---
oversampled_minority_data = np.concatenate((minority_data, synthetic_data_final), axis=0)
oversampled_minority_labels = np.ones(oversampled_minority_data.shape[0], dtype=int)

print(f"Shape of oversampled minority data: {oversampled_minority_data.shape}")

# --- (Optional) Combine with Majority Data ---
final_weather_data = np.concatenate((majority_data, oversampled_minority_data), axis=0)
final_labels = np.concatenate((labels[labels == 0], oversampled_minority_labels), axis=0)

# Optionally shuffle the final combined dataset
shuffle_indices = np.random.permutation(len(final_labels))
final_weather_data_2 = final_weather_data[shuffle_indices]
final_labels_2 = final_labels[shuffle_indices]

print(f"Shape of final balanced data: {final_weather_data_2.shape}")
print(f"Shape of final balanced labels: {final_labels_2.shape}")
print(f"Final class distribution: {np.bincount(final_labels_2)}")

# --- Now you can use 'final_weather_data' and 'final_labels' for your model training ---
```

```python
merged_df["event_type"].value_counts()
```

```python
plt.figure(figsize=(12, 6))
plt.plot(merged_df[merged_df["extreme"] == 1].iloc[0]["prev_72h_wind_gusts_10m"], label='Synth
```

```python
# crerate a line chart
import matplotlib.pyplot as plt

# Create a line chart for the first feature of the first synthetic sample
plt.figure(figsize=(12, 6))
plt.plot(oversampled_minority_data[5, 18, :], label='Synthetic Sample 1 - Feature 1')
```

Data inspection

```python
meteo_cols
```

```python
np.stack(merged_df[meteo_cols].values.tolist(), axis=0).shape
```

```python
merged_df.info()
```

```python
merged_df["event_type"].value_counts()
```

```python
merged_df_copy = merged_df[merged_df["event_type"].isin(["Flash Flood", "Flood","Heavy Rain"])
```

```python
merged_df_copy["prev_72h_temperature_2m"]
```

```python
merged_df_copy[merged_df_copy["extreme"] == 1].sample(1).iloc[0]
```

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Conv1D, MaxPooling1D, Dropout,
                                     LSTM, Dense, Input)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.metrics import Precision, Recall, AUC
import tensorflow as tf
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential, Model # Import Model for Functional API
from tensorflow.keras.layers import (
    Input, Conv1D, MaxPooling1D, BatchNormalization, Activation,
    LSTM, Bidirectional, Dropout, Dense, LeakyReLU
)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
# Import relevant metrics for classification, especially if data is imbalanced
from tensorflow.keras.metrics import Precision, Recall, AUC
# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# --- Data Preparation ---
# Assume merged_df and meteo_cols are defined earlier in your code.
# For multi-class, we now use the "event_type" column.

# Encode the categorical labels to integer values
le = LabelEncoder()
y_encoded = le.fit_transform(merged_df["event_type"].values)
# Number of classes (should be 10 based on provided counts)
num_classes = len(le.classes_)

# Convert integer labels to one-hot encoded format
y = tf.keras.utils.to_categorical(y_encoded, num_classes=num_classes)

# Prepare features (assuming meteo_cols contains your meteorological data)
# Here we simulate stacking the data; adjust if your actual data preprocessing is different.
X = np.stack(merged_df[meteo_cols].values.tolist(), axis=0)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y_encoded
)

# Scale the data
scaler = StandardScaler()
```

```
# Reshape for scaling: combine timesteps and features
X_train = X_train.reshape(-1, X_train.shape[-1])
X_test = X_test.reshape(-1, X_test.shape[-1])

# Fit and transform
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Reshape back to original structure: (samples, num_features, num_timesteps)
# Note: Here we assume the original shape was (samples, num_features, num_timesteps).
# Adjust reshaping if your data dimensions differ.
X_train = X_train.reshape(-1, 27, 72)
X_test = X_test.reshape(-1, 27, 72)

# Print shapes to verify correctness (should print something like (samples, 27, 72) and (sampl
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)

# --- Reshape Input for Model ---
# Rearranging shape to (samples, timesteps, features), here timesteps=72 and features=27.
X_train = X_train.transpose(0, 2, 1)
X_test  = X_test.transpose(0, 2, 1)

num_timesteps = X_train.shape[1]  # should be 72
num_features  = X_train.shape[2]  # should be 27
input_layer = Input(shape=(num_timesteps, num_features))
# --- CNN Feature Extraction Block ---
# Conv Block 1
x = Conv1D(filters=64, kernel_size=3, padding='same')(input_layer)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.01)(x) # Using LeakyReLU instead of ReLU
x = MaxPooling1D(pool_size=2)(x)
x = Dropout(0.3)(x) # Increased dropout slightly

# Conv Block 2
x = Conv1D(filters=128, kernel_size=5, padding='same')(x) # Larger kernel, more filters
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.01)(x)
# Optional: Add another MaxPooling here if needed, depends on sequence length
# x = MaxPooling1D(pool_size=2)(x)
x = Dropout(0.3)(x)

# Conv Block 3
x = Conv1D(filters=128, kernel_size=7, padding='same')(x) # Even larger kernel
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.01)(x)
x = Dropout(0.3)(x)

# --- LSTM Sequence Processing Block ---
# Using Bidirectional LSTM adds complexity and captures dependencies in both directions
# Stacked BiLSTM layers
x = Bidirectional(LSTM(128, return_sequences=True))(x) # return_sequences=True for stacking
x = Dropout(0.4)(x) # Higher dropout for recurrent layers

x = Bidirectional(LSTM(64))(x) # Last LSTM layer doesn't need return_sequences=True
```

```
x = Dropout(0.4)(x)

# --- Dense Classification Block ---
# Intermediate Dense layer
x = Dense(64)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.01)(x)
x = Dropout(0.5)(x) # Higher dropout before final layer

# Output layer
output_layer = Dense(3, activation='softmax')(x)

# Create the Model
model = Model(inputs=input_layer, outputs=output_layer)

# Compile the model with potentially more informative metrics
# Consider a lower learning rate for potentially more complex models
optimizer = Adam(learning_rate=1e-4, clipnorm=1.0)
model.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy',
                       Precision(name='precision'),
                       Recall(name='recall')]) # AUC is good for binary classification

model.summary()

# --- Callbacks ---
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6)

# --- Class Weights ---
# Calculate class weights using the original integer labels (y_encoded)
counts = np.bincount(y_encoded[train_test_split(range(len(y_encoded)),
                                                 test_size=0.2, random_state=42, stratify=y_en
# Ensure we compute weights for all classes
if len(counts) == num_classes:
    class_weight = {i: (1 / count) * (len(y_train)) / num_classes for i, count in enumerate(co
    print("Calculated Class Weights:", class_weight)
else:
    class_weight = None
    print("Mismatch in class counts; skipping class weight computation.")

# --- Train the Model ---
epochs = 50
batch_size = 32

history = model.fit(X_train, y_train,
                    epochs=epochs,
                    batch_size=batch_size,
                    validation_split=0.2,
                    callbacks=[early_stopping, reduce_lr],
                    class_weight=class_weight,
                    verbose=1)
```

```
# --- Evaluate the Model ---
print("\nEvaluating model on test set:")
```

```
results = model.evaluate(X_test, y_test, batch_size=batch_size, verbose=1)
print(f"Test Loss: {results[0]:.4f}")
print(f"Test Accuracy: {results[1]:.4f}")
print(f"Test Precision: {results[2]:.4f}")
print(f"Test Recall: {results[3]:.4f}")
```

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Conv1D, MaxPooling1D, Dropout,
                                     LSTM, Dense, Input)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.metrics import Precision, Recall, AUC
import tensorflow as tf
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential, Model # Import Model for Functional API
from tensorflow.keras.layers import (
    Input, Conv1D, MaxPooling1D, BatchNormalization, Activation,
    LSTM, Bidirectional, Dropout, Dense, LeakyReLU
)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
# Import relevant metrics for classification, especially if data is imbalanced
from tensorflow.keras.metrics import Precision, Recall, AUC
# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# --- Data Preparation ---
# Assume merged_df and meteo_cols are defined earlier in your code.
# For multi-class, we now use the "event_type" column.

# Encode the categorical labels to integer values
le = LabelEncoder()
y_encoded = le.fit_transform(merged_df["event_type"].values)
# Number of classes (should be 10 based on provided counts)
num_classes = len(le.classes_)

# Convert integer labels to one-hot encoded format
y = tf.keras.utils.to_categorical(y_encoded, num_classes=num_classes)

# Prepare features (assuming meteo_cols contains your meteorological data)
# Here we simulate stacking the data; adjust if your actual data preprocessing is different.
X = np.stack(merged_df[meteo_cols].values.tolist(), axis=0)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y_encoded
)

# Scale the data
```

```
scaler = StandardScaler()

# Reshape for scaling: combine timesteps and features
X_train = X_train.reshape(-1, X_train.shape[-1])
X_test = X_test.reshape(-1, X_test.shape[-1])

# Fit and transform
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Reshape back to original structure: (samples, num_features, num_timesteps)
# Note: Here we assume the original shape was (samples, num_features, num_timesteps).
# Adjust reshaping if your data dimensions differ.
X_train = X_train.reshape(-1, 27, 72)
X_test = X_test.reshape(-1, 27, 72)

# Print shapes to verify correctness (should print something like (samples, 27, 72) and (sampl
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)

# --- Reshape Input for Model ---
# Rearranging shape to (samples, timesteps, features), here timesteps=72 and features=27.
X_train = X_train.transpose(0, 2, 1)
X_test  = X_test.transpose(0, 2, 1)

num_timesteps = X_train.shape[1]  # should be 72
num_features  = X_train.shape[2]  # should be 27
input_layer = Input(shape=(num_timesteps, num_features))
# --- CNN Feature Extraction Block ---
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv1D, BatchNormalization, LeakyReLU, MaxPooling1D
from tensorflow.keras.optimizers import Adam



# --- Conv Block 1 ---
conv1 = Conv1D(filters=64, kernel_size=3, padding='same')(input_layer)
conv1 = BatchNormalization()(conv1)
conv1 = LeakyReLU(alpha=0.01)(conv1)  # using LeakyReLU instead of ReLU
conv1 = MaxPooling1D(pool_size=2)(conv1)
conv1 = Dropout(0.3)(conv1)

# --- Conv Block 2 ---
conv2 = Conv1D(filters=128, kernel_size=5, padding='same')(conv1)  # larger kernel, more filte
conv2 = BatchNormalization()(conv2)
conv2 = LeakyReLU(alpha=0.01)(conv2)
conv2 = Dropout(0.3)(conv2)

# --- Residual Connection ---
# Adjust conv1 dimensions to match conv2 via 1x1 convolution
shortcut = Conv1D(filters=128, kernel_size=1, padding='same')(conv1)
shortcut = BatchNormalization()(shortcut)
res = Add()([conv2, shortcut])

# --- Conv Block 3 ---
```

```
conv3 = Conv1D(filters=128, kernel_size=7, padding='same')(res)  # even larger kernel
conv3 = BatchNormalization()(conv3)
conv3 = LeakyReLU(alpha=0.01)(conv3)
conv3 = Dropout(0.3)(conv3)

# --- LSTM Sequence Processing Block ---
# Stacked Bidirectional LSTM with return_sequences=True to retain the full sequence output
x = Bidirectional(LSTM(128, return_sequences=True))(conv3)
x = Dropout(0.4)(x)
x = Bidirectional(LSTM(64, return_sequences=True))(x)
x = Dropout(0.4)(x)

# --- Attention Mechanism ---
# Using Keras built-in Attention layer for self-attention over the LSTM outputs.
# It takes [query, value] where both are the same in self-attention.
attn_out = Attention()([x, x])
# Global average pooling aggregates the attended sequence into a fixed-size vector.
x = GlobalAveragePooling1D()(attn_out)

# --- Dense Classification Block ---
x = Dense(64)(x)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.01)(x)
x = Dropout(0.5)(x)

# --- Output Layer ---
output_layer = Dense(3, activation='softmax')(x)

# --- Build and Compile the Model ---
model_1 = Model(inputs=input_layer, outputs=output_layer)

optimizer = Adam(learning_rate=1e-4, clipnorm=1.0)
model_1.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy',
                       tf.keras.metrics.Precision(name='precision'),
                       tf.keras.metrics.Recall(name='recall')])

model_1.summary()

# --- Callbacks ---
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6)

# --- Class Weights ---
# Calculate class weights using the original integer labels (y_encoded)
counts = np.bincount(y_encoded[train_test_split(range(len(y_encoded)),
                                                 test_size=0.2, random_state=42, stratify=y_en
# Ensure we compute weights for all classes
if len(counts) == num_classes:
    class_weight = {i: (1 / count) * (len(y_train)) / num_classes for i, count in enumerate(co
    print("Calculated Class Weights:", class_weight)
else:
    class_weight = None
    print("Mismatch in class counts; skipping class weight computation.")
```

```python
# --- Train the Model ---
epochs = 50
batch_size = 32

history_1 = model_1.fit(X_train, y_train,
                        epochs=epochs,
                        batch_size=batch_size,
                        validation_split=0.2,
                        callbacks=[early_stopping, reduce_lr],
                        class_weight=class_weight,
                        verbose=1)
```

```python
merged_df = merged_df[merged_df["event_type"].isin(["Flash Flood", "Flood","Heavy Rain"])].cop
```

```python
merged_df["extreme"].value_counts()
```

```python
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import RandomOverSampler



# Initialize the RandomOverSampler
oversample = RandomOverSampler(sampling_strategy='minority', random_state=42)

# Apply the oversampling to the dataset
X_resampled, y_resampled = oversample.fit_resample(merged_df.drop(columns=["extreme"]), merged

# Display class distribution after oversampling
print("Class distribution after oversampling:", Counter(y_resampled))
```

```python
import pandas as pd
import numpy as np

# Example datetime string
dt_str = "2012-10-29 12:30:00"
dt = pd.to_datetime(dt_str)

# Extract features
year = dt.year          # 2012
month = dt.month        # 10
day = dt.day            # 29
hour = dt.hour          # 12
minute = dt.minute      # 30
day_of_week = dt.dayofweek  # 0=Monday, 6=Sunday, for example

# Encode the hour cyclically
hour_rad = 2 * np.pi * hour / 24
sin_hour = np.sin(hour_rad)
cos_hour = np.cos(hour_rad)

print(f"Year: {year}, Month: {month}, Day: {day}, Day of Week: {day_of_week}")
print(f"Hour: {hour}, Sin(hour): {sin_hour:.2f}, Cos(hour): {cos_hour:.2f}")
```

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Conv1D, MaxPooling1D, Dropout,
                                     LSTM, Dense, Input)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.metrics import Precision, Recall, AUC
import tensorflow as tf
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential, Model # Import Model for Functional API
from tensorflow.keras.layers import (
    Input, Conv1D, MaxPooling1D, BatchNormalization, Activation,
    LSTM, Bidirectional, Dropout, Dense, LeakyReLU
)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
# Import relevant metrics for classification, especially if data is imbalanced
from tensorflow.keras.metrics import Precision, Recall, AUC
# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# --- Data Preparation ---
# Assume merged_df and meteo_cols are defined earlier in your code.
# For multi-class, we now use the "event_type" column.

# Encode the categorical labels to integer values
le = LabelEncoder()
y_encoded = le.fit_transform(merged_df["event_type"].values)
# Number of classes (should be 10 based on provided counts)
num_classes = len(le.classes_)

# Convert integer labels to one-hot encoded format
y = tf.keras.utils.to_categorical(y_encoded, num_classes=num_classes)
y_bin = merged_df["extreme"].values # Binary labels for extreme events
# Prepare features (assuming meteo_cols contains your meteorological data)
# Here we simulate stacking the data; adjust if your actual data preprocessing is different.


# Split the data into training and testing sets
X_train, X_test, y_train, y_test, y_train_bin, y_test_bin = train_test_split(
    merged_df[meteo_cols], y,y_bin, test_size=0.2, random_state=42, stratify=y_bin
)


X_train, X_validation, y_train_bin, y_validation_bin = train_test_split(
    X_train,y_train_bin, test_size=0.2, random_state=42, stratify=y_train_bin
)
from collections import Counter
from sklearn.datasets import make_classification
```

```
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
# Apply the oversampling to the dataset
oversample = RandomOverSampler(sampling_strategy='minority', random_state=42)
X_resampled, y_resampled = oversample.fit_resample(X_train,y_train_bin)


X_resampled = np.stack(X_resampled[meteo_cols].values.tolist(), axis=0)
X_test = np.stack(X_test[meteo_cols].values.tolist(), axis=0)
X_validation = np.stack(X_validation[meteo_cols].values.tolist(), axis=0)
# Initialize the RandomOverSampler

print("peecentage of positice instance in training data: ", sum(y_resampled)/len(y_resampled))
print("percentage of positice instance in test data: ", sum(y_test_bin)/len(y_test_bin))
print("percentage of positice instance in validation data: ", sum(y_validation_bin)/len(y_vali

print("y resampled shape:", y_resampled.shape)
# Display class distribution after oversampling
print("Class distribution after oversampling:", Counter(y_resampled))
# Scale the data
scaler = StandardScaler()

# Reshape for scaling: combine timesteps and features
X_resampled = X_resampled.reshape(-1, X_resampled.shape[-1])
X_test = X_test.reshape(-1, X_test.shape[-1])
X_validation = X_validation.reshape(-1, X_validation.shape[-1])
# Fit and transform
print("X resampled shape before scaling:", X_resampled.shape)
X_resampled = scaler.fit_transform(X_resampled)
X_test = scaler.transform(X_test)
X_validation = scaler.transform(X_validation)
# Reshape back to original structure: (samples, num_features, num_timesteps)
# Note: Here we assume the original shape was (samples, num_features, num_timesteps).
# Adjust reshaping if your data dimensions differ.
X_resampled = X_resampled.reshape(-1, 27, 72)
X_test = X_test.reshape(-1, 27, 72)
X_validation = X_validation.reshape(-1, 27, 72)
# Print shapes to verify correctness (should print something like (samples, 27, 72) and (sampl
print("X_train shape:", X_resampled.shape)
print("y_train shape:", y_resampled.shape)

# --- Reshape Input for Model ---
# Rearranging shape to (samples, timesteps, features), here timesteps=72 and features=27.
X_resampled = X_resampled.transpose(0, 2, 1)
X_test  = X_test.transpose(0, 2, 1)
X_validation = X_validation.transpose(0, 2, 1)
num_timesteps = X_resampled.shape[1]  # should be 72
num_features  = X_resampled.shape[2]  # should be 27
input_layer = Input(shape=(num_timesteps, num_features))
# --- CNN Feature Extraction Block ---
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv1D, BatchNormalization, LeakyReLU, MaxPooling1D
from tensorflow.keras.optimizers import Adam
```

```
# --- Shared Conv Block 1 ---
conv1 = Conv1D(filters=64, kernel_size=3, padding='same')(input_layer)
conv1 = BatchNormalization()(conv1)
conv1 = LeakyReLU(alpha=0.01)(conv1)
conv1 = MaxPooling1D(pool_size=2)(conv1)
conv1 = Dropout(0.3)(conv1)

# --- Shared Conv Block 2 ---
conv2 = Conv1D(filters=128, kernel_size=5, padding='same')(conv1)
conv2 = BatchNormalization()(conv2)
conv2 = LeakyReLU(alpha=0.01)(conv2)
conv2 = Dropout(0.3)(conv2)

# --- Residual Connection from Conv Block 1 ---
shortcut = Conv1D(filters=128, kernel_size=1, padding='same')(conv1)
shortcut = BatchNormalization()(shortcut)
res = Add()([conv2, shortcut])

# --- Shared Conv Block 3 ---
conv3 = Conv1D(filters=128, kernel_size=7, padding='same')(res)
conv3 = BatchNormalization()(conv3)
conv3 = LeakyReLU(alpha=0.01)(conv3)
conv3 = Dropout(0.3)(conv3)

# --- Shared LSTM Sequence Processing Block ---
x = Bidirectional(LSTM(128, return_sequences=True))(conv3)
x = Dropout(0.4)(x)
x = Bidirectional(LSTM(64, return_sequences=True))(x)
x = Dropout(0.4)(x)

# --- Shared Attention Mechanism ---
# Using built-in Keras Attention layer (self-attention)
attn_out = Attention()([x, x])
# Aggregate attended features into a fixed-size vector
shared_features = GlobalAveragePooling1D()(attn_out)

# --- Event Type Classification Head ---
event_branch = Dense(64)(shared_features)
event_branch = BatchNormalization()(event_branch)
event_branch = LeakyReLU(alpha=0.01)(event_branch)
event_branch = Dropout(0.5)(event_branch)
event_output = Dense(3, activation='softmax', name='event_type')(event_branch)

# --- Extreme Event Classification Head ---
extreme_branch = Dense(64)(shared_features)
extreme_branch = BatchNormalization()(extreme_branch)
extreme_branch = LeakyReLU(alpha=0.01)(extreme_branch)
extreme_branch = Dropout(0.5)(extreme_branch)
extreme_output = Dense(1, activation='sigmoid', name='extreme')(extreme_branch)

# --- Create and Compile the Multi-task Model ---
model = Model(inputs=input_layer, outputs=extreme_output)
```

```
optimizer = Adam(learning_rate=1e-4, clipnorm=1.0)
# model.compile(optimizer=optimizer,
#               loss={'event_type': 'categorical_crossentropy', 'extreme': 'binary_crossentrop
#               metrics={'event_type': ['accuracy'], 'extreme': [tf.keras.metrics.Precision(),

model.compile(optimizer=optimizer,
              loss={'extreme': 'binary_crossentropy'},
              metrics={'extreme': [tf.keras.metrics.Accuracy(name='accuracy'), tf.keras.metric

model.summary()

# --- Callbacks ---
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6)

# --- Class Weights ---
from sklearn.utils.class_weight import compute_class_weight
class_weights = compute_class_weight(
    'balanced',   # This option adjusts weights inversely proportional to class frequencies ir
    classes=np.unique(y_resampled),  # List of unique class labels
    y=y_resampled  # Your target variable (e.g., labels)
)

# Convert class_weights to a dictionary format
class_weight = dict(zip(np.unique(y_resampled), class_weights))
# --- Train the Model ---
epochs = 50
batch_size = 32
print("y resampe shape", y_resampled.shape)
history_1 = model.fit(X_resampled,  y_resampled,
                    epochs=epochs,
                    batch_size=batch_size,
                    validation_split=0.2,
                    callbacks=[early_stopping, reduce_lr],
                    validation_data=(X_validation, y_validation_bin),
                    class_weight=class_weight,
                    verbose=1)
```

```
print(model.predict(X_test))
```

```
# After training, find optimal threshold for F1 score instead of default 0.5
def find_optimal_threshold(model, x_val, y_val):
    y_pred_prob = model.predict(x_val)

    best_f1 = 0
    best_threshold = 0.5

    for threshold in np.arange(0.1, 0.9, 0.01):
        y_pred = (y_pred_prob > threshold).astype(int)
        precision = tf.keras.metrics.Precision()(y_val, y_pred).numpy()
        recall = tf.keras.metrics.Recall()(y_val, y_pred).numpy()

        if precision + recall > 0:  # Avoid division by zero
            f1 = 2 * (precision * recall) / (precision + recall)
```

```
            if f1 > best_f1:
                best_f1 = f1
                best_threshold = threshold

    print(f"Optimal threshold: {best_threshold}, F1 score: {best_f1:.4f}")
    return best_threshold

# Use the optimal threshold for final predictions
optimal_threshold = find_optimal_threshold(model, X_validation, y_validation_bin)
y_pred = (model.predict(X_test) > optimal_threshold).astype(int)

precision = precision_score(y_test_bin, y_pred)
recall = recall_score(y_test_bin, y_pred)
f1 = f1_score(y_test_bin, y_pred)
roc_auc = roc_auc_score(y_test_bin, y_probs)
accuracy = accuracy_score(y_test_bin, y_pred)

# 4. Display the metrics
print(f"Test Accuracy: {accuracy:.4f}")
print(f"Test Precision: {precision:.4f}")
print(f"Test Recall: {recall:.4f}")
print(f"Test F1-Score: {f1:.4f}")
print(f"Test ROC AUC: {roc_auc:.4f}")

# 5. Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_test_bin, y_probs)

# 6. Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()

# 7. Compute confusion matrix
cm = confusion_matrix(y_test_bin, y_pred)

# 8. Plot confusion matrix using Seaborn heatmap
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted Negative', 'Predict
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.xlabel('Predicted Labels')
plt.ylabel('Actual Labels')
plt.title('Confusion Matrix')
plt.show()
```

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import (precision_score, recall_score, f1_score, roc_auc_score,
                             roc_curve, confusion_matrix, accuracy_score)
from sklearn.preprocessing import LabelBinarizer
```

```python
# Assuming 'model' is your trained model and 'X_test', 'y_test' are your test data and labels

# 1. Obtain predicted probabilities for the positive class
y_probs = model.predict(X_test).flatten() # Adjust indexing if your model has multiple outputs

# 2. Binarize the true labels if they are not already in binary forma

# 3. Compute evaluation metrics
y_pred = (y_probs > 0.5).astype(int)
precision = precision_score(y_test_bin, y_pred)
recall = recall_score(y_test_bin, y_pred)
f1 = f1_score(y_test_bin, y_pred)
roc_auc = roc_auc_score(y_test_bin, y_probs)
accuracy = accuracy_score(y_test_bin, y_pred)

# 4. Display the metrics
print(f"Test Accuracy: {accuracy:.4f}")
print(f"Test Precision: {precision:.4f}")
print(f"Test Recall: {recall:.4f}")
print(f"Test F1-Score: {f1:.4f}")
print(f"Test ROC AUC: {roc_auc:.4f}")

# 5. Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_test_bin, y_probs)

# 6. Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()

# 7. Compute confusion matrix
cm = confusion_matrix(y_test_bin, y_pred)

# 8. Plot confusion matrix using Seaborn heatmap
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted Negative', 'Predict
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.xlabel('Predicted Labels')
plt.ylabel('Actual Labels')
plt.title('Confusion Matrix')
plt.show()
```

```python
precision = results[2]
recall = results[3]

# Calculate F1 Score
f1_score = 2 * (precision * recall) / (precision + recall)

# Display the F1 Score
print(f"Test F1 Score: {f1_score:.4f}")
```

```
import tensorflow as tf
from tensorflow.keras.layers import (Input, Conv1D, BatchNormalization, LeakyReLU, ReLU, Add,
                                     Dense, GlobalAveragePooling1D, LayerNormalization, MultiH
                                     Layer, Embedding, Input, SpatialDropout1D)
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import math
import tensorflow as tf
from tensorflow.keras.layers import (Input, Conv1D, BatchNormalization, LeakyReLU, ReLU, Add,
                                     Dense, GlobalAveragePooling1D, LayerNormalization, MultiH
                                     Layer, Embedding, Input, SpatialDropout1D)
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import math

# --- Shared Configuration ---
INPUT_SHAPE = (72, 27) # (time_steps, num_features)
NUM_CLASSES = 1        # Binary classification for 'extreme'
LEARNING_RATE = 1e-4
CLIPNORM = 1.0

# --- Metrics (same as your last model) ---
METRICS = [
    tf.keras.metrics.Accuracy(name='accuracy'),
    tf.keras.metrics.Precision(name='precision'),
    tf.keras.metrics.Recall(name='recall'),
    tf.keras.metrics.AUC(name='roc_auc'),
    tf.keras.metrics.AUC(curve='PR', name='pr_auc')
]
# --- TCN Residual Block ---
def TCNBlock(input_tensor, filters, kernel_size, dilation_rate, dropout_rate=0.2, activation=
    """A single TCN residual block."""
    # Ensure causal padding
    conv1 = Conv1D(filters=filters, kernel_size=kernel_size, dilation_rate=dilation_rate,
                   padding='causal', kernel_initializer='he_normal')(input_tensor)
    # Using LayerNorm instead of BatchNorm is sometimes preferred in sequence models
    norm1 = LayerNormalization()(conv1)
    # Using ReLU is more common in TCN literature than LeakyReLU
    act1 = ReLU()(norm1) if activation == 'relu' else LeakyReLU(alpha=0.01)(norm1)
    drop1 = SpatialDropout1D(dropout_rate)(act1) # SpatialDropout drops entire feature maps

    conv2 = Conv1D(filters=filters, kernel_size=kernel_size, dilation_rate=dilation_rate,
                   padding='causal', kernel_initializer='he_normal')(drop1)
    norm2 = LayerNormalization()(conv2)
    act2 = ReLU()(norm2) if activation == 'relu' else LeakyReLU(alpha=0.01)(norm2)
    drop2 = SpatialDropout1D(dropout_rate)(act2)

    # Residual connection (project input if number of filters differs)
    if input_tensor.shape[-1] != filters:
        shortcut = Conv1D(filters=filters, kernel_size=1, padding='same', kernel_initializer=
    else:
        shortcut = input_tensor
```

```
    res = Add()([shortcut, drop2])
    # Optional: Activation after Add (some TCN versions do, some don't)
    # res = ReLU()(res) if activation == 'relu' else LeakyReLU(alpha=0.01)(res)
    return res

# --- Build the TCN Model ---
def build_tcn_model(input_shape, num_classes, num_tcn_blocks=4, filters=64,
                    kernel_size=3, dropout_rate=0.2, final_dropout=0.5):
    """Builds the TCN model."""
    input_layer = Input(shape=input_shape)
    x = input_layer

    # Stack TCN Blocks with increasing dilation
    for i in range(num_tcn_blocks):
        dilation_rate = 2**i
        # Increase filters in deeper layers (optional)
        current_filters = filters * (2**(i // 2)) # e.g., 64, 64, 128, 128
        x = TCNBlock(x, filters=current_filters, kernel_size=kernel_size,
                     dilation_rate=dilation_rate, dropout_rate=dropout_rate)

    # Aggregate features over time
    x = GlobalAveragePooling1D()(x) # Or use x = x[:, -1, :] to take the last time step output

    # Classification Head
    x = Dense(64)(x)
    x = LayerNormalization()(x) # Using LayerNorm here too
    x = LeakyReLU(alpha=0.01)(x)
    x = Dropout(final_dropout)(x)
    output_layer = Dense(num_classes, activation='sigmoid', name='extreme')(x) # Sigmoid for b

    model = Model(inputs=input_layer, outputs=output_layer)
    return model

# --- Create and Compile TCN Model ---
print("--- Building TCN Model ---")
tcn_model = build_tcn_model(
    input_shape=INPUT_SHAPE,
    num_classes=NUM_CLASSES,
    num_tcn_blocks=4, # Adjust number of blocks
    filters=64,       # Starting filters
    kernel_size=3,    # Kernel size for TCN blocks
    dropout_rate=0.2, # Dropout within TCN blocks
    final_dropout=0.5 # Dropout in the final classification head
)

optimizer_tcn = Adam(learning_rate=LEARNING_RATE, clipnorm=CLIPNORM)
tcn_model.compile(optimizer=optimizer_tcn,
                  loss='binary_crossentropy',
                  metrics=METRICS)

tcn_model.summary()
# --- Positional Encoding ---
# Necessary for Transformers to understand sequence order
class PositionalEncoding(Layer):
    def __init__(self, position, d_model):
        super(PositionalEncoding, self).__init__()
```

```
        self.pos_encoding = self.positional_encoding(position, d_model)

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'position': self.pos_encoding.shape[1],
             'd_model': self.pos_encoding.shape[2],
         })
        return config

    def positional_encoding(self, position, d_model):
        angle_rads = self.get_angles(np.arange(position)[:, np.newaxis],
                                     np.arange(d_model)[np.newaxis, :],
                                     d_model)
        # apply sin to even indices in the array; 2i
        angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
        # apply cos to odd indices in the array; 2i+1
        angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
        pos_encoding = angle_rads[np.newaxis, ...]
        return tf.cast(pos_encoding, dtype=tf.float32)

    def get_angles(self, pos, i, d_model):
        angle_rates = 1 / np.power(10000, (2 * (i // 2)) / np.float32(d_model))
        return pos * angle_rates

    def call(self, inputs):
        seq_len = tf.shape(inputs)[1]
        # Ensure positional encoding is not longer than the input sequence
        return inputs + self.pos_encoding[:, :seq_len, :]


# --- Transformer Encoder Block ---
class TransformerBlock(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, dropout_rate=0.1):
        super(TransformerBlock, self).__init__()
        self.att = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim // num_heads) # A
        # Feed Forward Network
        self.ffn = tf.keras.Sequential(
            [Dense(ff_dim, activation="relu"), Dense(embed_dim),] # Use ReLU common in Transfo
        )
        # Layer Normalization
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        # Dropout
        self.dropout1 = Dropout(dropout_rate)
        self.dropout2 = Dropout(dropout_rate)

    def get_config(self):
        config = super().get_config().copy()
        config.update({
             'embed_dim': self.ffn.layers[1].units, # Get embed_dim from dense layer output
             'num_heads': self.att.num_heads,
             'ff_dim': self.ffn.layers[0].units, # Get ff_dim from dense layer units
             'dropout_rate': self.dropout1.rate # Assuming dropout rates are the same
         })
        return config
```

```python
    def call(self, inputs, training=False):
        # Multi-Head Attention
        attn_output = self.att(inputs, inputs) # Self-attention
        # Dropout and Residual Connection
        out1 = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + out1)
        # Feed Forward Network
        ffn_output = self.ffn(out1)
        # Dropout and Residual Connection
        out2 = self.dropout2(ffn_output, training=training)
        out2 = self.layernorm2(out1 + out2)
        return out2


# --- Build the Transformer Model ---
def build_transformer_model(input_shape, num_classes, num_transformer_blocks,
                            embed_dim, num_heads, ff_dim, dropout_rate=0.1,
                            final_dropout=0.5, mlp_units=64):
    """Builds the Transformer Encoder model."""
    time_steps = input_shape[0]
    num_features = input_shape[1]
    inputs = Input(shape=input_shape)

    # 1. Input Embedding/Projection
    # Project input features (27) to the embedding dimension (e.g., 128)
    # Using Conv1D allows mixing features across channels at each time step
    x = Conv1D(filters=embed_dim, kernel_size=1, activation='relu')(inputs)
    # Or use TimeDistributed Dense:
    # x = TimeDistributed(Dense(embed_dim, activation='relu'))(inputs)

    # 2. Add Positional Encoding
    x = PositionalEncoding(position=time_steps, d_model=embed_dim)(x)
    x = Dropout(dropout_rate)(x) # Dropout after embedding + pos encoding

    # 3. Stack Transformer Blocks
    for _ in range(num_transformer_blocks):
        x = TransformerBlock(embed_dim, num_heads, ff_dim, dropout_rate)(x)

    # 4. Pooling / Aggregation
    # Global Average Pooling aggregates across the time dimension
    x = GlobalAveragePooling1D()(x)
    # Optional: Use the output of the first token ([CLS] token style) if you design it that wo

    # 5. Classification Head
    x = Dense(mlp_units)(x)
    x = LayerNormalization()(x) # Normalize before activation
    x = LeakyReLU(alpha=0.01)(x) # Or ReLU
    x = Dropout(final_dropout)(x)
    outputs = Dense(num_classes, activation="sigmoid", name="extreme")(x) # Sigmoid for binary

    model = Model(inputs=inputs, outputs=outputs)
    return model

# --- Create and Compile Transformer Model ---
print("\n--- Building Transformer Model ---")
```

```python
transformer_model = build_transformer_model(
    input_shape=INPUT_SHAPE,
    num_classes=NUM_CLASSES,
    num_transformer_blocks=3, # Number of stacked encoder blocks (tune this)
    embed_dim=128,            # Embedding dimension (vector size for each time step)
    num_heads=8,              # Number of attention heads (must divide embed_dim)
    ff_dim=128,               # Hidden layer size in feed forward network (e.g., 4*embed_dim c
    dropout_rate=0.1,         # Dropout rate within Transformer blocks
    final_dropout=0.5,        # Dropout in the final classification head
    mlp_units=64              # Units in the classification head's hidden layer
)

optimizer_transformer = Adam(learning_rate=LEARNING_RATE, clipnorm=CLIPNORM)
transformer_model.compile(optimizer=optimizer_transformer,
                          loss='binary_crossentropy',
                          metrics=METRICS)

transformer_model.summary()
```

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Conv1D, MaxPooling1D, Dropout,
                                     LSTM, Dense, Input)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.metrics import Precision, Recall, AUC
import tensorflow as tf
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential, Model # Import Model for Functional API
from tensorflow.keras.layers import (
    Input, Conv1D, MaxPooling1D, BatchNormalization, Activation,
    LSTM, Bidirectional, Dropout, Dense, LeakyReLU
)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
# Import relevant metrics for classification, especially if data is imbalanced
from tensorflow.keras.metrics import Precision, Recall, AUC
# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# --- Data Preparation ---
# Assume merged_df and meteo_cols are defined earlier in your code.
# For multi-class, we now use the "event_type" column.

# Encode the categorical labels to integer values
le = LabelEncoder()
y_encoded = le.fit_transform(merged_df["event_type"].values)
# Number of classes (should be 10 based on provided counts)
num_classes = len(le.classes_)
```

```
# Convert integer labels to one-hot encoded format
y = tf.keras.utils.to_categorical(y_encoded, num_classes=num_classes)
y_bin = merged_df["extreme"].values # Binary labels for extreme events
# Prepare features (assuming meteo_cols contains your meteorological data)
# Here we simulate stacking the data; adjust if your actual data preprocessing is different.


# Split the data into training and testing sets
X_train, X_test, y_train, y_test, y_train_bin, y_test_bin = train_test_split(
    merged_df[meteo_cols], y,y_bin, test_size=0.2, random_state=42, stratify=y_bin
)


X_train, X_validation, y_train_bin, y_validation_bin = train_test_split(
    X_train,y_train_bin, test_size=0.2, random_state=42, stratify=y_train_bin
)
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
# Apply the oversampling to the dataset
oversample = RandomOverSampler(sampling_strategy='minority', random_state=42)
X_resampled, y_resampled = oversample.fit_resample(X_train,y_train_bin)


X_resampled = np.stack(X_resampled[meteo_cols].values.tolist(), axis=0)
X_test = np.stack(X_test[meteo_cols].values.tolist(), axis=0)
X_validation = np.stack(X_validation[meteo_cols].values.tolist(), axis=0)
# Initialize the RandomOverSampler

print("peecentage of positice instance in training data: ", sum(y_resampled)/len(y_resampled))
print("percentage of positice instance in test data: ", sum(y_test_bin)/len(y_test_bin))
print("percentage of positice instance in validation data: ", sum(y_validation_bin)/len(y_vali

print("y resampled shape:", y_resampled.shape)
# Display class distribution after oversampling
print("Class distribution after oversampling:", Counter(y_resampled))
# Scale the data
scaler = StandardScaler()

# Reshape for scaling: combine timesteps and features
X_resampled = X_resampled.reshape(-1, X_resampled.shape[-1])
X_test = X_test.reshape(-1, X_test.shape[-1])
X_validation = X_validation.reshape(-1, X_validation.shape[-1])
# Fit and transform
print("X resampled shape before scaling:", X_resampled.shape)
X_resampled = scaler.fit_transform(X_resampled)
X_test = scaler.transform(X_test)
X_validation = scaler.transform(X_validation)
# Reshape back to original structure: (samples, num_features, num_timesteps)
# Note: Here we assume the original shape was (samples, num_features, num_timesteps).
# Adjust reshaping if your data dimensions differ.
X_resampled = X_resampled.reshape(-1, 27, 72)
X_test = X_test.reshape(-1, 27, 72)
X_validation = X_validation.reshape(-1, 27, 72)
# Print shapes to verify correctness (should print something like (samples, 27, 72) and (sampl
```

```python
print("X_train shape:", X_resampled.shape)
print("y_train shape:", y_resampled.shape)

# --- Reshape Input for Model ---
# Rearranging shape to (samples, timesteps, features), here timesteps=72 and features=27.
X_resampled = X_resampled.transpose(0, 2, 1)
X_test  = X_test.transpose(0, 2, 1)
X_validation = X_validation.transpose(0, 2, 1)
num_timesteps = X_resampled.shape[1]  # should be 72
num_features  = X_resampled.shape[2]  # should be 27
input_layer = Input(shape=(num_timesteps, num_features))
# --- CNN Feature Extraction Block ---
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv1D, BatchNormalization, LeakyReLU, MaxPooling1D
from tensorflow.keras.optimizers import Adam


# --- Callbacks ---
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6)

# --- Class Weights ---
from sklearn.utils.class_weight import compute_class_weight
class_weights = compute_class_weight(
    'balanced',   # This option adjusts weights inversely proportional to class frequencies in
    classes=np.unique(y_resampled),  # List of unique class labels
    y=y_resampled  # Your target variable (e.g., labels)
)

# Convert class_weights to a dictionary format
class_weight = dict(zip(np.unique(y_resampled), class_weights))
# --- Train the Model ---
epochs = 50
batch_size = 32
print("y resampe shape", y_resampled.shape)
history_1 = simple_transformer_model.fit(X_resampled,  y_resampled,
                    epochs=epochs,
                    batch_size=batch_size,
                    validation_split=0.2,
                    callbacks=[early_stopping, reduce_lr],
                    validation_data=(X_validation, y_validation_bin),
                    class_weight=class_weight,
                    verbose=1)
```

```python
import tensorflow as tf
from tensorflow.keras.layers import (Input, LayerNormalization, MultiHeadAttention, Dense,
                                     Dropout, GlobalAveragePooling1D, Layer, Conv1D, Add)
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import math

# --- Configuration ---
INPUT_SHAPE = (72, 27) # (time_steps, num_features)
NUM_CLASSES = 1        # Binary classification for 'extreme'
```

```
LEARNING_RATE = 1e-4
CLIPNORM = 1.0

# --- Transformer Hyperparameters (Tune these!) ---
EMBED_DIM = 64          # Dimension of the transformer embeddings (feature vector per time ste
NUM_HEADS = 4           # Number of attention heads (must divide EMBED_DIM)
FF_DIM = 64            # Hidden layer size in the Feed Forward network inside the block
NUM_TRANSFORMER_BLOCKS = 2 # How many blocks to stack
DROPOUT_RATE = 0.1      # Dropout rate within transformer blocks
FINAL_DROPOUT = 0.2     # Dropout rate in the final classification head
MLP_UNITS = 64          # Units in the final classification head's hidden layer

# --- Metrics ---
METRICS = [
    tf.keras.metrics.Accuracy(name='accuracy'),
    tf.keras.metrics.Precision(name='precision'),
    tf.keras.metrics.Recall(name='recall'),
    tf.keras.metrics.AUC(name='roc_auc'),
    tf.keras.metrics.AUC(curve='PR', name='pr_auc')
]

# --- Positional Encoding Layer ---
class PositionalEncoding(Layer):
    def __init__(self, position, d_model, **kwargs):
        super(PositionalEncoding, self).__init__(**kwargs)
        self.position = position
        self.d_model = d_model
        self.pos_encoding = self.build_positional_encoding(position, d_model)

    def get_config(self):
        config = super().get_config()
        config.update({
            'position': self.position,
            'd_model': self.d_model,
        })
        return config

    def build_positional_encoding(self, position, d_model):
        angle_rads = self.get_angles(np.arange(position)[:, np.newaxis],
                                     np.arange(d_model)[np.newaxis, :],
                                     d_model)
        # apply sin to even indices in the array; 2i
        angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
        # apply cos to odd indices in the array; 2i+1
        angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
        pos_encoding = angle_rads[np.newaxis, ...]
        return tf.cast(pos_encoding, dtype=tf.float32)

    def get_angles(self, pos, i, d_model):
        angle_rates = 1 / np.power(10000, (2 * (i // 2)) / np.float32(d_model))
        return pos * angle_rates

    def call(self, inputs):
        seq_len = tf.shape(inputs)[1]
        # Add positional encoding (make sure it's not longer than input seq)
        return inputs + self.pos_encoding[:, :seq_len, :]
```

```python
# --- Transformer Encoder Block Layer ---
class TransformerBlock(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, dropout_rate=0.1, **kwargs):
        super(TransformerBlock, self).__init__(**kwargs)
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.ff_dim = ff_dim
        self.dropout_rate = dropout_rate

        # Ensure key_dim is calculated appropriately if not default
        # key_dim = embed_dim // num_heads
        # if embed_dim % num_heads != 0:
        #     raise ValueError(f"embed_dim ({embed_dim}) must be divisible by num_heads ({num_

        self.att = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim) # Using embed_di
        self.ffn = tf.keras.Sequential(
            [Dense(ff_dim, activation="relu"), Dense(embed_dim),]
        )
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(dropout_rate)
        self.dropout2 = Dropout(dropout_rate)

    def get_config(self):
        config = super().get_config()
        config.update({
            'embed_dim': self.embed_dim,
            'num_heads': self.num_heads,
            'ff_dim': self.ff_dim,
            'dropout_rate': self.dropout_rate
        })
        return config

    def call(self, inputs, training=False):
        # Multi-Head Self-Attention
        attn_output = self.att(query=inputs, value=inputs, key=inputs, training=training) # Se
        # Dropout & Residual Connection 1
        out1 = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + out1) # Add & Norm
        # Feed Forward Network
        ffn_output = self.ffn(out1, training=training)
        # Dropout & Residual Connection 2
        out2 = self.dropout2(ffn_output, training=training)
        out2 = self.layernorm2(out1 + out2) # Add & Norm
        return out2

# --- Build the Simple Transformer Model ---
def build_simple_transformer(input_shape, num_classes, num_transformer_blocks,
                             embed_dim, num_heads, ff_dim, dropout_rate,
                             final_dropout, mlp_units):
    """Builds a simple stacked Transformer Encoder model."""
    time_steps = input_shape[0]
    num_features = input_shape[1]
    inputs = Input(shape=input_shape)
```

```
    # 1. Input Projection
    # Use a Conv1D layer to project the 27 features to the embedding dimension (embed_dim)
    # This allows the model to learn a representation for each time step.
    x = Conv1D(filters=embed_dim, kernel_size=1, padding='same', activation='relu')(inputs)
    # Alternatively, use a Dense layer applied to each time step:
    # from tensorflow.keras.layers import TimeDistributed
    # x = TimeDistributed(Dense(embed_dim, activation='relu'))(inputs)
    x = LayerNormalization(epsilon=1e-6)(x) # Normalize after projection
    x = Dropout(dropout_rate)(x)

    # 2. Add Positional Encoding
    x = PositionalEncoding(position=time_steps, d_model=embed_dim)(x)

    # 3. Stack Transformer Blocks
    for _ in range(num_transformer_blocks):
        x = TransformerBlock(embed_dim, num_heads, ff_dim, dropout_rate)(x)

    # 4. Pooling / Aggregation
    # Global Average Pooling aggregates the output sequence across time
    x = GlobalAveragePooling1D()(x)

    # 5. Final Classification Head
    x = Dropout(final_dropout)(x)
    x = Dense(mlp_units, activation='relu')(x) # Hidden dense layer
    x = Dropout(final_dropout)(x)
    outputs = Dense(num_classes, activation="sigmoid", name="extreme")(x) # Output layer (Sigm

    model = Model(inputs=inputs, outputs=outputs)
    return model

# --- Create and Compile the Model ---
print("--- Building Simple Transformer Model ---")
simple_transformer_model = build_simple_transformer(
    input_shape=INPUT_SHAPE,
    num_classes=NUM_CLASSES,
    num_transformer_blocks=NUM_TRANSFORMER_BLOCKS,
    embed_dim=EMBED_DIM,
    num_heads=NUM_HEADS,
    ff_dim=FF_DIM,
    dropout_rate=DROPOUT_RATE,
    final_dropout=FINAL_DROPOUT,
    mlp_units=MLP_UNITS
)

optimizer = Adam(learning_rate=LEARNING_RATE, clipnorm=CLIPNORM)
simple_transformer_model.compile(optimizer=optimizer,
                                 loss='binary_crossentropy',
                                 metrics=METRICS)

simple_transformer_model.summary()

# --- How to Train (Example) ---
# Prepare your data X_train (shape: N, 72, 27), y_train (shape: N, 1)
# Prepare validation data X_val, y_val

# history = simple_transformer_model.fit(
```

```
#     X_train, y_train,
#     batch_size=64,      # Tune this
#     epochs=50,         # Tune this
#     validation_data=(X_val, y_val),
#     # Add class_weight={0: w0, 1: w1} if you have class imbalance
#     # callbacks=[tf.keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True), .
# )
```

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import (precision_score, recall_score, f1_score, roc_auc_score,
                             roc_curve, confusion_matrix, accuracy_score)
from sklearn.preprocessing import LabelBinarizer

# Assuming 'model' is your trained model and 'X_test', 'y_test' are your test data and labels

# 1. Obtain predicted probabilities for the positive class
y_probs = model.predict(X_test).flatten() # Adjust indexing if your model has multiple outputs

# 2. Binarize the true labels if they are not already in binary forma

# 3. Compute evaluation metrics
y_pred = (y_probs > 0.5).astype(int)
precision = precision_score(y_test_bin, y_pred)
recall = recall_score(y_test_bin, y_pred)
f1 = f1_score(y_test_bin, y_pred)
roc_auc = roc_auc_score(y_test_bin, y_probs)
accuracy = accuracy_score(y_test_bin, y_pred)

# 4. Display the metrics
print(f"Test Accuracy: {accuracy:.4f}")
print(f"Test Precision: {precision:.4f}")
print(f"Test Recall: {recall:.4f}")
print(f"Test F1-Score: {f1:.4f}")
print(f"Test ROC AUC: {roc_auc:.4f}")

# 5. Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_test_bin, y_probs)

# 6. Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()

# 7. Compute confusion matrix
cm = confusion_matrix(y_test_bin, y_pred)

# 8. Plot confusion matrix using Seaborn heatmap
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted Negative', 'Predict
```

```
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.xlabel('Predicted Labels')
plt.ylabel('Actual Labels')
plt.title('Confusion Matrix')
plt.show()
```

adding location and time information

```
merged_df.info()
```

```
import numpy as np
import pandas as pd
from collections import Counter
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from imblearn.over_sampling import RandomOverSampler
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import (
    Input, Conv1D, MaxPooling1D, BatchNormalization, LeakyReLU,
    Dropout, Add, Bidirectional, LSTM, GlobalAveragePooling1D,
    Attention, Dense, Concatenate
)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.metrics import Accuracy, Precision, Recall, AUC

# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# ========================
# --- Data Preparation ---
# ========================

# Assume merged_df and meteo_cols are defined.
# merged_df should have:
#   - meteo_cols: Your meteorological sequence data (each row: a list structure that you stack
#   - "lat" and "lon": Location features.
#   - "begin_datetime": A datetime string column (e.g., "2012-10-29 12:30:00").
#   - "event_type": A categorical event column.
#   - "extreme": A binary target indicating extreme events.

# Convert begin_datetime to datetime and extract time features
merged_df['begin_datetime'] = pd.to_datetime(merged_df['begin_date_time'])
merged_df['hour'] = merged_df['begin_datetime'].dt.hour
# Cyclical encoding for hour (24-hour periodicity)
merged_df['sin_hour'] = np.sin(2 * np.pi * merged_df['hour'] / 24)
merged_df['cos_hour'] = np.cos(2 * np.pi * merged_df['hour'] / 24)
merged_df['dayofweek'] = merged_df['begin_datetime'].dt.dayofweek  # 0=Mon, ... 6=Sun

# List of extra features (time & location)
extra_cols = ['sin_hour', 'cos_hour', 'dayofweek', 'begin_lat', 'begin_lon']
extra_data = merged_df[extra_cols]
```

```
# --- Prepare Targets ---
# For the multi-class event type (if needed)
le = LabelEncoder()
y_encoded = le.fit_transform(merged_df["event_type"].values)
num_classes = len(le.classes_)
y_cat = tf.keras.utils.to_categorical(y_encoded, num_classes=num_classes)

# For extreme event (binary target)
y_bin = merged_df["extreme"].values

# --- Prepare Meteorological Sequence Data ---
# meteo_cols is assumed to be a list of column names in merged_df with your sequence informati
# Each row in merged_df[meteo_cols] is expected to be a list (or similar structure) that will
X_seq_raw = merged_df[meteo_cols]

# ----------------------
# --- Data Splitting ---
# ----------------------
# Split based on indices; here we use the extreme label for stratification.
X_seq_train, X_seq_test, y_train_cat, y_test_cat, y_train_bin, y_test_bin = train_test_split(
    X_seq_raw, y_cat, y_bin, test_size=0.2, random_state=42, stratify=y_bin
)

X_seq_train, X_seq_val, y_train_bin, y_val_bin = train_test_split(
    X_seq_train, y_train_bin, test_size=0.2, random_state=42, stratify=y_train_bin
)

# Also split extra features using the same indices as for the sequence data
X_extra_train = extra_data.loc[X_seq_train.index].copy()
X_extra_test  = extra_data.loc[X_seq_test.index].copy()
X_extra_val   = extra_data.loc[X_seq_val.index].copy()

# ---------------------------------------------------
# --- Combine Sequence and Extra Data for Oversample ---
# ---------------------------------------------------
# To ensure that oversampling picks the same rows for both branches,
# combine them into one DataFrame. We assume here that X_seq_train contains
# each row as a list (e.g., stored as an object in the DataFrame), so we will not modify it.
# Instead, add the extra features as additional columns.
X_train_combined = X_seq_train.copy()
for col in extra_cols:
    X_train_combined[col] = X_extra_train[col].values

# Apply oversampling on the combined DataFrame based on y_train_bin.
oversample = RandomOverSampler(sampling_strategy='minority', random_state=42)
X_train_comb_res, y_train_bin_res = oversample.fit_resample(X_train_combined, y_train_bin)

print("Percentage of positive instances in training data: ", np.mean(y_train_bin_res))
print("Percentage of positive instances in test data: ", np.mean(y_test_bin))
print("Percentage of positive instances in validation data: ", np.mean(y_val_bin))
print("Class distribution after oversampling:", Counter(y_train_bin_res))

# ---------------------------------------------------
# --- Separate Combined Data Back into Two Parts ---
# ---------------------------------------------------
```

```python
# For the sequence part, we retrieve the original meteo_cols that are stored as lists.
X_seq_train_res = X_train_comb_res[meteo_cols]

# For the extra features, retrieve the extra_cols.
X_extra_train_res = X_train_comb_res[extra_cols]

# ----------------------------------------------
# --- Process Sequence Data (Stack & Scale) ---
# ----------------------------------------------
# Stack the sequence data: each element in X_seq_train_res is assumed to be a list.
X_train_seq_arr = np.stack(X_seq_train_res.values.tolist(), axis=0)
X_test_seq_arr = np.stack(X_seq_test[meteo_cols].values.tolist(), axis=0)
X_val_seq_arr = np.stack(X_seq_val[meteo_cols].values.tolist(), axis=0)

# For scaling, reshape so each timestep is treated independently.
train_shape = X_train_seq_arr.shape  # (samples, num_features, timesteps)
X_train_seq_flat = X_train_seq_arr.reshape(-1, train_shape[-1])
X_test_seq_flat = X_test_seq_arr.reshape(-1, X_test_seq_arr.shape[-1])
X_val_seq_flat = X_val_seq_arr.reshape(-1, X_val_seq_arr.shape[-1])

scaler_seq = StandardScaler()
X_train_seq_flat = scaler_seq.fit_transform(X_train_seq_flat)
X_test_seq_flat = scaler_seq.transform(X_test_seq_flat)
X_val_seq_flat = scaler_seq.transform(X_val_seq_flat)

# Reshape back to original structure and transpose to (samples, timesteps, features)
X_train_seq_arr = X_train_seq_flat.reshape(train_shape).transpose(0, 2, 1)
X_test_seq_arr = X_test_seq_flat.reshape(X_test_seq_arr.shape).transpose(0, 2, 1)
X_val_seq_arr = X_val_seq_flat.reshape(X_val_seq_arr.shape).transpose(0, 2, 1)

num_timesteps = X_train_seq_arr.shape[1]      # e.g., 72 timesteps
num_sequence_features = X_train_seq_arr.shape[2]  # e.g., 27 features

# -----------------------------------------
# --- Process Extra Features (Scale) ---
# -----------------------------------------
scaler_extra = StandardScaler()
X_train_extra_arr = scaler_extra.fit_transform(X_extra_train_res.values)
X_test_extra_arr  = scaler_extra.transform(X_extra_test.values)
X_val_extra_arr   = scaler_extra.transform(X_extra_val.values)
num_extra_features = X_train_extra_arr.shape[1]  # Should be 5 here

print("X_train_seq shape:", X_train_seq_arr.shape)
print("X_train_extra shape:", X_train_extra_arr.shape)
print("y_train_bin shape:", np.array(y_train_bin_res).shape)

# ========================
# --- Build the Model ---
# ========================

# Sequence Input (meteorological time series)
seq_input = Input(shape=(num_timesteps, num_sequence_features), name='timeseries_input')

# --- CNN Feature Extraction for Sequence Data ---
# Shared Conv Block 1
conv1 = Conv1D(filters=64, kernel_size=3, padding='same')(seq_input)
```

```python
conv1 = BatchNormalization()(conv1)
conv1 = LeakyReLU(negative_slope=0.01)(conv1)
conv1 = MaxPooling1D(pool_size=2)(conv1)
conv1 = Dropout(0.3)(conv1)

# Shared Conv Block 2
conv2 = Conv1D(filters=128, kernel_size=5, padding='same')(conv1)
conv2 = BatchNormalization()(conv2)
conv2 = LeakyReLU(negative_slope=0.01)(conv2)
conv2 = Dropout(0.3)(conv2)

# Residual Connection from Conv Block 1
shortcut = Conv1D(filters=128, kernel_size=1, padding='same')(conv1)
shortcut = BatchNormalization()(shortcut)
res = Add()([conv2, shortcut])

# Shared Conv Block 3
conv3 = Conv1D(filters=128, kernel_size=7, padding='same')(res)
conv3 = BatchNormalization()(conv3)
conv3 = LeakyReLU(negative_slope=0.01)(conv3)
conv3 = Dropout(0.3)(conv3)

# Shared LSTM Sequence Processing Block
x = Bidirectional(LSTM(128, return_sequences=True))(conv3)
x = Dropout(0.4)(x)
x = Bidirectional(LSTM(64, return_sequences=True))(x)
x = Dropout(0.4)(x)

# Shared Attention Mechanism
attn_out = Attention()([x, x])
shared_features = GlobalAveragePooling1D()(attn_out)

# --- Extra Features Branch (Time & Location) ---
extra_input = Input(shape=(num_extra_features,), name='extra_input')
extra_branch = Dense(32, activation='relu')(extra_input)
extra_branch = BatchNormalization()(extra_branch)
extra_branch = Dropout(0.3)(extra_branch)

# --- Concatenate Both Branches ---
combined_features = Concatenate()([shared_features, extra_branch])

# --- Classification Head for Extreme Event Classification ---
extreme_branch = Dense(64)(combined_features)
extreme_branch = BatchNormalization()(extreme_branch)
extreme_branch = LeakyReLU(negative_slope=0.01)(extreme_branch)
extreme_branch = Dropout(0.5)(extreme_branch)
extreme_output = Dense(1, activation='sigmoid', name='extreme')(extreme_branch)

# --- Create and Compile the Model ---
model = Model(inputs=[seq_input, extra_input], outputs=extreme_output)
optimizer = Adam(learning_rate=1e-4, clipnorm=1.0)
model.compile(
    optimizer=optimizer,
    loss={'extreme': 'binary_crossentropy'},
    metrics={'extreme': [
        Accuracy(name='accuracy'),
```

```
        Precision(),
        Recall(),
        AUC(name='roc_auc'),
        AUC(curve='PR', name='pr_auc')
    ]}
)
model.summary()

# -------------------------------------
# --- Callbacks and Class Weight Setup
# -------------------------------------
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6)

from sklearn.utils.class_weight import compute_class_weight
class_weights = compute_class_weight(
    'balanced',
    classes=np.unique(y_train_bin_res),
    y=y_train_bin_res
)
class_weight_dict = dict(zip(np.unique(y_train_bin_res), class_weights))

# -----------------------
# --- Train the Model ---
# -----------------------
epochs = 50
batch_size = 32
history = model.fit(
    [X_train_seq_arr, X_train_extra_arr], y_train_bin_res,
    epochs=epochs,
    batch_size=batch_size,
    validation_data=([X_val_seq_arr, X_val_extra_arr], y_val_bin),
    callbacks=[early_stopping, reduce_lr],
    class_weight=class_weight_dict,
    verbose=1
)

# Optionally evaluate on test data:
results = model.evaluate([X_test_seq_arr, X_test_extra_arr], y_test_bin, verbose=1)
print("Test results:", results)
```

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import (precision_score, recall_score, f1_score, roc_auc_score,
                             roc_curve, confusion_matrix, accuracy_score)
from sklearn.preprocessing import LabelBinarizer

# Assuming 'model' is your trained model and 'X_test', 'y_test' are your test data and labels

# 1. Obtain predicted probabilities for the positive class
y_probs = model.predict([X_test_seq_arr, X_extra_test]).flatten() # Adjust indexing if your mo

# 2. Binarize the true labels if they are not already in binary forma

# 3. Compute evaluation metrics
```

```
y_pred = (y_probs > 0.5).astype(int)
precision = precision_score(y_test_bin, y_pred)
recall = recall_score(y_test_bin, y_pred)
f1 = f1_score(y_test_bin, y_pred)
roc_auc = roc_auc_score(y_test_bin, y_probs)
accuracy = accuracy_score(y_test_bin, y_pred)

# 4. Display the metrics
print(f"Test Accuracy: {accuracy:.4f}")
print(f"Test Precision: {precision:.4f}")
print(f"Test Recall: {recall:.4f}")
print(f"Test F1-Score: {f1:.4f}")
print(f"Test ROC AUC: {roc_auc:.4f}")

# 5. Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_test_bin, y_probs)

# 6. Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()

# 7. Compute confusion matrix
cm = confusion_matrix(y_test_bin, y_pred)

# 8. Plot confusion matrix using Seaborn heatmap
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted Negative', 'Predict
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.xlabel('Predicted Labels')
plt.ylabel('Actual Labels')
plt.title('Confusion Matrix')
plt.show()
```

Try Different deep learning models with above processed data at each step like:

1. Train simple neural networks wiht above vairation on the dataset(like with standardization, normalization, PCA, etc.)
2. Try LSTM, GRU, CNN-LSTM, Transformer with proper input features

```

```

Do all the above things with image feature extraction

```python
import ast
import tensorflow as tf
import pandas as pd

# Example: create a sample DataFrame with a column 'filenames'
data = {
    'filenames': [
        "['event0_Flood_IMERG_Precipitation_Rate_2012-10-29.png', 'event0_Flood_VIIRS_SNPP_Ice
    ]
}
df = pd.DataFrame(data)

# Base directory where your images are stored
base_image_path = "/path/to/your/images/"

# Function to convert the string representation to a Python list of filenames.
def parse_filenames(filenames_str):
    # Safely evaluate the string to get the list.
    return ast.literal_eval(filenames_str)

# Function to load and preprocess a single image.
def load_and_preprocess_image(image_path, image_size=(224, 224)):
    # Read the image from disk.
    image = tf.io.read_file(image_path)
    # Decode PNG (or use tf.image.decode_jpeg for jpg).
    image = tf.image.decode_png(image, channels=3)
    # Convert image to floats in range [0, 1].
    image = tf.image.convert_image_dtype(image, tf.float32)
    # Resize the image to the desired dimensions.
    image = tf.image.resize(image, image_size)
    return image

# Function to load all images from the list of filenames and stack them.
def load_images_from_list(filenames_str, base_path=base_image_path, image_size=(224, 224)):
    # Convert string list into actual list of filenames.
    filenames = parse_filenames(filenames_str)
    # Construct full paths and load each image.
    images = []
    for fname in filenames:
        full_path = base_path + fname  # Adjust joining if needed (os.path.join is safer)
        img = load_and_preprocess_image(full_path, image_size)
        images.append(img)
    # Stack into a single tensor with shape (num_images, height, width, channels)
    images_tensor = tf.stack(images)
    return images_tensor

# Test the image loading function on the first row.
example_filenames_str = df.loc[0, 'filenames']
images_tensor = load_images_from_list(example_filenames_str)
print("Shape of loaded images tensor:", images_tensor.shape)

# --------------------------------------------------
# Optional: Building a tf.data Pipeline
# --------------------------------------------------
def process_row(filenames_str):
```

```
    # Here, convert the string to a stacked image tensor.
    images_tensor = load_images_from_list(filenames_str)
    # You can further process images_tensor if needed
    # For example, you might want to reshape or merge different days into groups if necessary.
    return images_tensor

# Create a TensorFlow dataset from the 'filenames' column.
filenames_ds = tf.data.Dataset.from_tensor_slices(df['filenames'].tolist())

# Map the dataset to load images
images_ds = filenames_ds.map(lambda x: tf.py_function(func=process_row, inp=[x], Tout=tf.float

# For demonstration, iterate through the dataset and print the shape.
for image_tensor in images_ds.take(1):
    print("tf.data loaded images shape:", image_tensor.shape)
```

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Conv1D, MaxPooling1D, Dropout,
                                     LSTM, Dense, Input)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.metrics import Precision, Recall, AUC
import tensorflow as tf
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential, Model # Import Model for Functional API
from tensorflow.keras.layers import (
    Input, Conv1D, MaxPooling1D, BatchNormalization, Activation,
    LSTM, Bidirectional, Dropout, Dense, LeakyReLU
)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
# Import relevant metrics for classification, especially if data is imbalanced
from tensorflow.keras.metrics import Precision, Recall, AUC
# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# --- Data Preparation ---
# Assume merged_df and meteo_cols are defined earlier in your code.
# For multi-class, we now use the "event_type" column.

# Encode the categorical labels to integer values
le = LabelEncoder()
y_encoded = le.fit_transform(merged_df["event_type"].values)
# Number of classes (should be 10 based on provided counts)
num_classes = len(le.classes_)

# Convert integer labels to one-hot encoded format
y = tf.keras.utils.to_categorical(y_encoded, num_classes=num_classes)
y_bin = merged_df["extreme"].values # Binary labels for extreme events
```

```python
# Prepare features (assuming meteo_cols contains your meteorological data)
# Here we simulate stacking the data; adjust if your actual data preprocessing is different.


# Split the data into training and testing sets
X_train, X_test, y_train, y_test, y_train_bin, y_test_bin = train_test_split(
    merged_df[meteo_cols], y,y_bin, test_size=0.2, random_state=42, stratify=y_bin
)


X_train, X_validation, y_train_bin, y_validation_bin = train_test_split(
    X_train,y_train_bin, test_size=0.2, random_state=42, stratify=y_train_bin
)
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
# Apply the oversampling to the dataset
oversample = RandomOverSampler(sampling_strategy='minority', random_state=42)
X_resampled, y_resampled = oversample.fit_resample(X_train,y_train_bin)


X_resampled = np.stack(X_resampled[meteo_cols].values.tolist(), axis=0)
X_test = np.stack(X_test[meteo_cols].values.tolist(), axis=0)
X_validation = np.stack(X_validation[meteo_cols].values.tolist(), axis=0)
# Initialize the RandomOverSampler

print("peecentage of positice instance in training data: ", sum(y_resampled)/len(y_resampled)
print("percentage of positice instance in test data: ", sum(y_test_bin)/len(y_test_bin))
print("percentage of positice instance in validation data: ", sum(y_validation_bin)/len(y_vali

print("y resampled shape:", y_resampled.shape)
# Display class distribution after oversampling
print("Class distribution after oversampling:", Counter(y_resampled))
# Scale the data
scaler = StandardScaler()

# Reshape for scaling: combine timesteps and features
X_resampled = X_resampled.reshape(-1, X_resampled.shape[-1])
X_test = X_test.reshape(-1, X_test.shape[-1])
X_validation = X_validation.reshape(-1, X_validation.shape[-1])
# Fit and transform
print("X resampled shape before scaling:", X_resampled.shape)
X_resampled = scaler.fit_transform(X_resampled)
X_test = scaler.transform(X_test)
X_validation = scaler.transform(X_validation)
# Reshape back to original structure: (samples, num_features, num_timesteps)
# Note: Here we assume the original shape was (samples, num_features, num_timesteps).
# Adjust reshaping if your data dimensions differ.
X_resampled = X_resampled.reshape(-1, 27, 72)
X_test = X_test.reshape(-1, 27, 72)
X_validation = X_validation.reshape(-1, 27, 72)
# Print shapes to verify correctness (should print something like (samples, 27, 72) and (sampl
print("X_train shape:", X_resampled.shape)
print("y_train shape:", y_resampled.shape)
```

```
# --- Reshape Input for Model ---
# Rearranging shape to (samples, timesteps, features), here timesteps=72 and features=27.
X_resampled = X_resampled.transpose(0, 2, 1)
X_test  = X_test.transpose(0, 2, 1)
X_validation = X_validation.transpose(0, 2, 1)
num_timesteps = X_resampled.shape[1]  # should be 72
num_features  = X_resampled.shape[2]  # should be 27
input_layer = Input(shape=(num_timesteps, num_features))
# --- CNN Feature Extraction Block ---
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv1D, BatchNormalization, LeakyReLU, MaxPooling1D
from tensorflow.keras.optimizers import Adam







import tensorflow as tf
from tensorflow.keras.layers import Input, Conv1D, BatchNormalization, LeakyReLU, MaxPooling1D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

import tensorflow as tf
from tensorflow.keras.layers import (Input, Conv1D, Conv2D, BatchNormalization, LeakyReLU,
                                     MaxPooling1D, MaxPooling2D, Dropout, Add, Bidirectional,
                                     LSTM, Attention, GlobalAveragePooling1D, GlobalAveragePoo
                                     Dense, concatenate)
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

# ---------------------------
# Weather Data Branch
# ---------------------------
# For example, assume weather data is a sequence of 100 timesteps with 64 features.
weather_input = Input(shape=(100, 64), name='weather_input')

# Conv Block 1
w = Conv1D(filters=64, kernel_size=3, padding='same')(weather_input)
w = BatchNormalization()(w)
w = LeakyReLU(alpha=0.01)(w)
w = MaxPooling1D(pool_size=2)(w)
w = Dropout(0.3)(w)

# Conv Block 2
w = Conv1D(filters=128, kernel_size=5, padding='same')(w)
w = BatchNormalization()(w)
w = LeakyReLU(alpha=0.01)(w)
w = Dropout(0.3)(w)

# Residual Connection from Block 1
shortcut = Conv1D(filters=128, kernel_size=1, padding='same')(weather_input[:, :weather_input.
shortcut = BatchNormalization()(shortcut)
w = Add()([w, shortcut])
```

```
# Conv Block 3
w = Conv1D(filters=128, kernel_size=7, padding='same')(w)
w = BatchNormalization()(w)
w = LeakyReLU(alpha=0.01)(w)
w = Dropout(0.3)(w)

# LSTM and Attention
w = Bidirectional(LSTM(128, return_sequences=True))(w)
w = Dropout(0.4)(w)
w = Bidirectional(LSTM(64, return_sequences=True))(w)
w = Dropout(0.4)(w)
w = Attention()([w, w])
weather_features = GlobalAveragePooling1D()(w)

# ---------------------------
# Image Data Branch (Feature Extraction)
# ---------------------------
# For example, assume raw images are of shape (224, 224, 3)
image_input = Input(shape=(224, 224, 3), name='image_input')

# Custom CNN for feature extraction
i = Conv2D(32, kernel_size=(3,3), padding='same')(image_input)
i = BatchNormalization()(i)
i = LeakyReLU(alpha=0.01)(i)
i = MaxPooling2D(pool_size=(2,2))(i)
i = Dropout(0.3)(i)

i = Conv2D(64, kernel_size=(3,3), padding='same')(i)
i = BatchNormalization()(i)
i = LeakyReLU(alpha=0.01)(i)
i = MaxPooling2D(pool_size=(2,2))(i)
i = Dropout(0.3)(i)

i = Conv2D(128, kernel_size=(3,3), padding='same')(i)
i = BatchNormalization()(i)
i = LeakyReLU(alpha=0.01)(i)
i = MaxPooling2D(pool_size=(2,2))(i)
i = Dropout(0.3)(i)

# Global pooling to reduce to feature vector
i = GlobalAveragePooling2D()(i)
# Optional fully connected layer to further process image features
img_features = Dense(128, activation='relu')(i)
img_features = BatchNormalization()(img_features)
img_features = Dropout(0.4)(img_features)

# ---------------------------
# Merge Branches and Build Final Classification Head
# ---------------------------
# Merge the weather and image features
combined_features = concatenate([weather_features, img_features], name='combined_features')

# Fully connected layers for extreme event classification
x = Dense(64)(combined_features)
x = BatchNormalization()(x)
x = LeakyReLU(alpha=0.01)(x)
```

```
x = Dropout(0.5)(x)
extreme_output = Dense(1, activation='sigmoid', name='extreme')(x)

# Define the complete multi-modal model
model = Model(inputs=[weather_input, image_input], outputs=extreme_output)

# Compile the model with Adam optimizer and binary cross-entropy loss
optimizer = Adam(learning_rate=1e-4, clipnorm=1.0)
model.compile(optimizer=optimizer,
              loss='binary_crossentropy',
              metrics=[tf.keras.metrics.Accuracy(name='accuracy'),
                       tf.keras.metrics.Precision(),
                       tf.keras.metrics.Recall(),
                       tf.keras.metrics.AUC(name='roc_auc'),
                       tf.keras.metrics.AUC(curve='PR', name='pr_auc')])

# Display the model summary
model.summary()



# --- Callbacks ---
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6)

# --- Class Weights ---
from sklearn.utils.class_weight import compute_class_weight
class_weights = compute_class_weight(
    'balanced',   # This option adjusts weights inversely proportional to class frequencies ir
    classes=np.unique(y_resampled),  # List of unique class labels
    y=y_resampled  # Your target variable (e.g., labels)
)

# Convert class_weights to a dictionary format
class_weight = dict(zip(np.unique(y_resampled), class_weights))
# --- Train the Model ---
epochs = 50
batch_size = 32
print("y resampe shape", y_resampled.shape)
history_1 = model.fit(X_resampled,  y_resampled,
                    epochs=epochs,
                    batch_size=batch_size,
                    validation_split=0.2,
                    callbacks=[early_stopping, reduce_lr],
                    validation_data=(X_validation, y_validation_bin),
                    class_weight=class_weight,
                    verbose=1)
```

Do all the above things with finetuning a good pretrained image model

Do all the above things without finetuning but with CNN model from scratch

Try Forecasting + Binary classification and anomaly detection

```python
import pandas as pd

merged_df = pd.read_csv(r"C:\Personal\Capstone\merged_df_copy.csv")
```

```python
merged_df['extreme'].value_counts()
```

```python
meteo_cols = [col for col in merged_df.columns if col.startswith("prev_72h")]
meteo_cols.remove("prev_72h_weather")
meteo_cols.remove("prev_72h_weather_code")
meteo_cols.remove("prev_72h_snow_depth")
meteo_cols.remove("prev_72h_snowfall")
```

```python
import numpy as np
from tqdm import tqdm
import ast
for col in tqdm(meteo_cols):
    merged_df[col] = merged_df[col].apply(lambda x: np.array(ast.literal_eval(x)) if isinstanc
```

```python
metadata = pd.read_csv(r"C:\Personal\Capstone\satellite_images\satellite_images_metadata.csv")
```

```python
filenames = []

# Ensure the DataFrame has enough rows
if len(merged_df) > 7000:
    for i in range(7000, len(merged_df)):
        temp = []
        event_id = merged_df["event_id"].iloc[i]

        # Check if event_id exists in metadata
        if event_id in metadata["event_id"].values:
            for j in metadata[metadata["event_id"] == event_id]["image_filename"].values:

                temp.append(j)
        else:
            print(f"event_id {event_id} not found in metadata")


        filenames.append(temp)
else:
    print("merged_df has fewer than 7000 rows")
```

```
merged_df["filenames"].iloc[7000:] = filenames
```

```
merged_df["filenames"].iloc[:7000] = merged_df["filenames"].iloc[:7000].apply(ast.literal_eval
```

```
merged_df["filenames"] =  merged_df["filenames"].apply(lambda x: [f for f in x if "MODIS_Terra
```

```
merged_df["filenames"] = merged_df["filenames"].apply(lambda x: x[::-1])
```

```
merged_df[]
```

```
import ast
import tensorflow as tf
import pandas as pd


# Base directory where your images are stored
base_image_path = [r"C:\Personal\Capstone\satellite_images\\", r"C:\Personal\Capstone project\

# Function to convert the string representation to a Python list of filenames.
def parse_filenames(filenames_str):
    # Safely evaluate the string to get the list.
    return filenames_str

# Function to load and preprocess a single image.
def load_and_preprocess_image(filename, folder_paths, image_size=(224, 224)):
    # Read the image from disk.
    for path in folder_paths:
        try:
            final_path = path + filename
            image = tf.io.read_file(final_path)
            # Decode PNG (or use tf.image.decode_jpeg for jpg).
            image = tf.image.decode_png(image, channels=3)
            # Convert image to floats in range [0, 1].
            image = tf.image.convert_image_dtype(image, tf.float32)
            # Resize the image to the desired dimensions.
            image = tf.image.resize(image, image_size)
            return image
        except Exception as e:
            continue
    # Function to load all images from the list of filenames and stack them.
def load_images_from_list(filenames_str, base_path=base_image_path, image_size=(64, 64)):
    # Convert string list into actual list of filenames.
    filenames = parse_filenames(filenames_str)
    # Construct full paths and load each image.
    images = []
    for fname in filenames:

        img = load_and_preprocess_image(fname, base_path, image_size)
        images.append(img)
    # Stack into a single tensor with shape (num_images, height, width, channels)
    images_tensor = tf.stack(images)
    return images_tensor

# Test the image loading function on the first row.
```

```
example_filenames_str = merged_df["filenames"].iloc[0]
images_tensor = load_images_from_list(example_filenames_str)
print("Shape of loaded images tensor:", images_tensor.shape)
```

```
import pandas as pd
import numpy as np
from tqdm import tqdm
import gc

def batch_process_images(df, batch_size=50, checkpoint_interval=10):
    """
    Process images in batches and store directly in dataframe to avoid OOM errors
    """
    # Create a copy to avoid modifying the original during processing
    result_df = df.copy()

    # Initialize the column with None values
    result_df["images_loaded"] = None

    # Calculate number of batches
    num_batches = (len(df) + batch_size - 1) // batch_size

    for batch_idx in tqdm(range(num_batches), desc="Processing batches"):
        start_idx = batch_idx * batch_size
        end_idx = min((batch_idx + 1) * batch_size, len(df))

        # Process just this batch of rows
        batch_slice = slice(start_idx, end_idx)

        # Apply the function to just this batch
        result_df.loc[batch_slice, "images_loaded"] = (
            df.loc[batch_slice, "filenames"].apply(load_images_from_list)
        )

        # Save checkpoint periodically
        if (batch_idx + 1) % checkpoint_interval == 0 or batch_idx == num_batches - 1:
            checkpoint_file = f'df_with_images/merged_df_checkpoint_{batch_idx+1}of{num_batche
            result_df.to_pickle(checkpoint_file)
            print(f"Checkpoint saved: {checkpoint_file}")

        # Force garbage collection to free memory
        gc.collect()

    return result_df

# Use this function instead of direct apply
merged_df = batch_process_images(merged_df, batch_size=2000, checkpoint_interval=4)
```

```
merged_df["filenames"] = merged_df["filenames"].apply(lambda x: [f for f in x if "MODIS_Terra_
```

```
import numpy as np
print(np.__version__)
```

```
import pandas as pd
```

```python
print(pd.__version__)
```

```python
!pip show numpy
```

```python

```

```python
%pip install nbconvert
```

```python

```